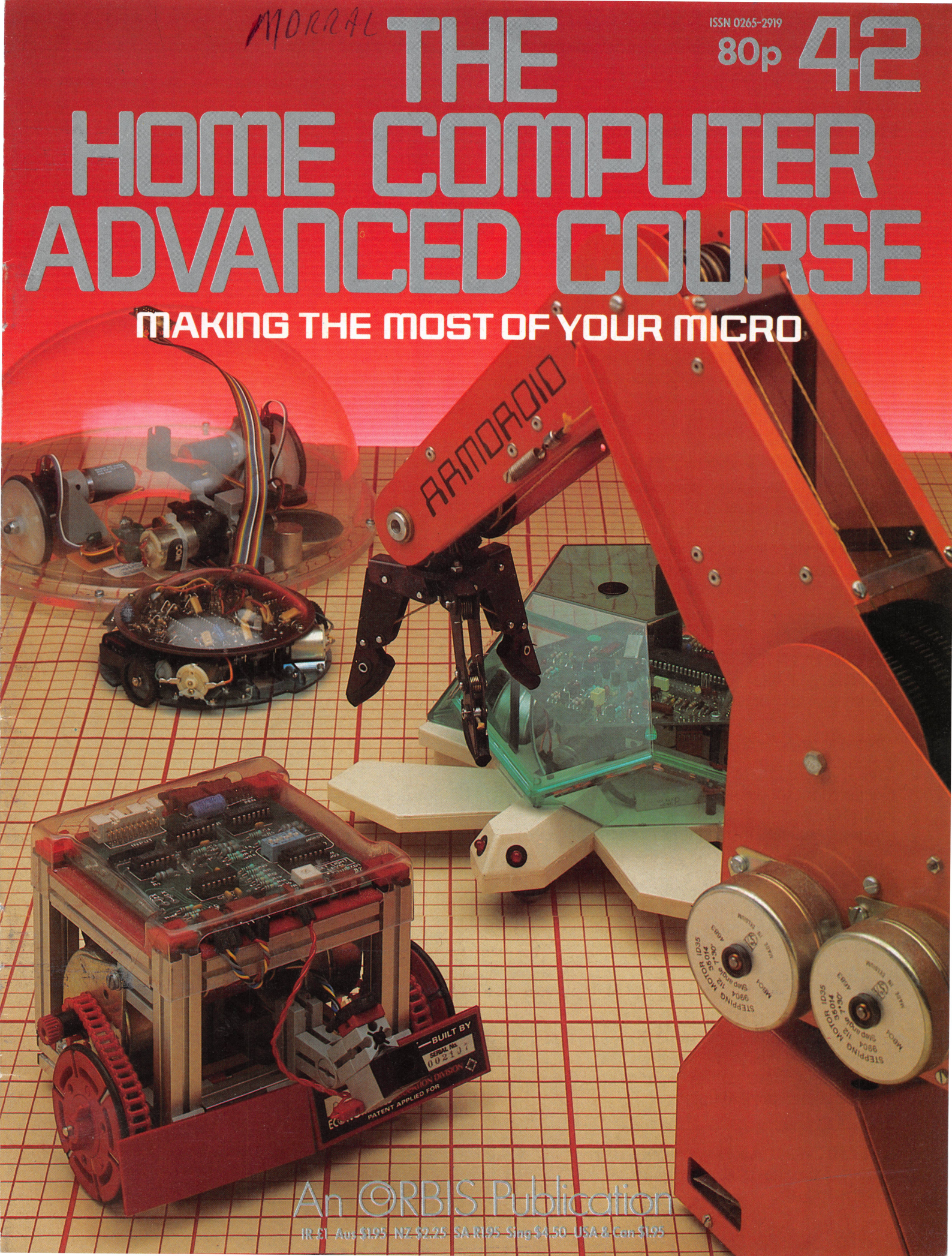
MORRAL

ISSN 0265-2919

80p 42

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



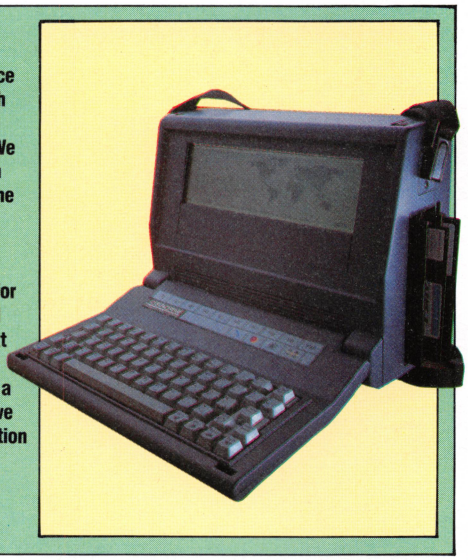
An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **Osborne changed the face of business computing with their original portable computer, the Osborne 1. We examine their chances of a repeat performance with the IBM-compatible Osborne Encore.**
- **If you think vertical software is another name for jumpsuits, you should read our new series starting next week.**
- **In the first instalment of a new machine code series we begin an in-depth examination of computer operating systems.**



QUIZ

1) Where might you find an unexplained foot switch?
2) What is a modem icon?
3) When is half a loaf as good as a whole one?

Answers To Last Week's Quiz

1) The new feature on the Spectrum+ is the reset button, positioned on the left-hand side of the casing.
2) We are using stepper motors in our robot project. Because they move in discrete steps, they can be moved with great precision.
3) To access an array in memory we would normally expect to use indexed addressing.
4) An equation processor enables you to define variables in a flexible format and can solve for any variable, while spreadsheets have a more rigidly defined construction.

Editor Mike Wesley; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Stephen Malone; **Sub Editor** Steve Mann; **Consultant Editor** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellar, Mike Curtis, Steve Colwill, Chris Naylor, Henry Budgett; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: Aus$1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZS2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY CHRIS STEVENS — ROBOTS COURTESY OF ROBOTICS WORKSHOP

FITTING THE BILL

As we continue our series on the topic of robotics, we examine some of the products sold commercially under the name 'robot' to see if they fill the role expected of them, and if they fit our carefully constructed definition of the term.

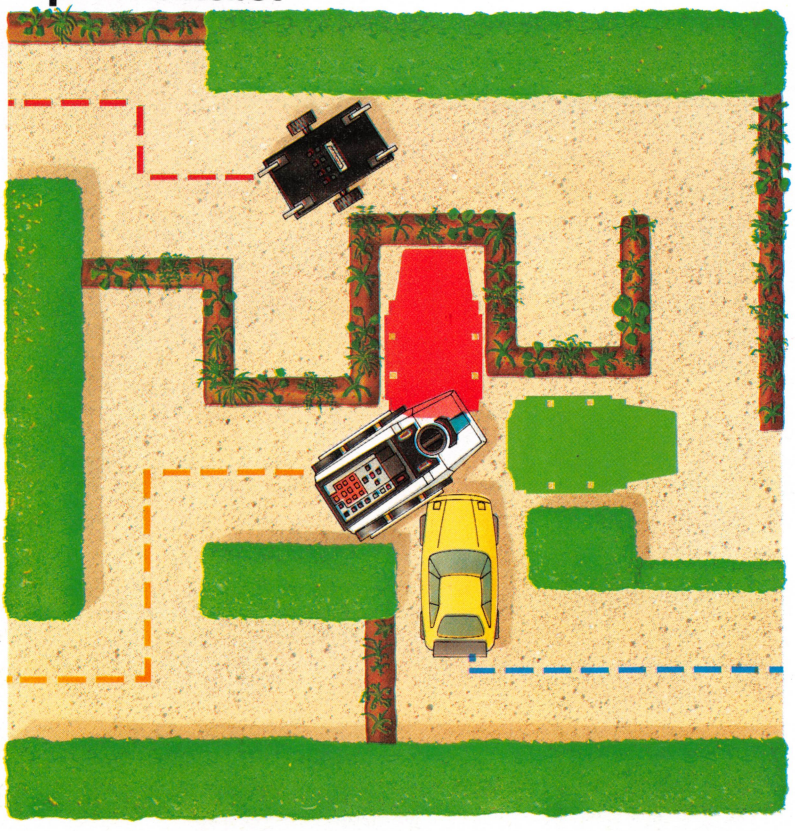
Up to this point in the Robotics series, we have dealt primarily with theoretical considerations of robot design and operation. In practice, many of the concepts discussed have not been implemented, or are restricted by a lack of funding, intricate mechanical parts, and/or intelligent software. Existing robots, whether they are intended for home or industrial use, tend to fall short of what we have come to expect of robots over the years. Sensors exist to make a robot see, hear, or feel, but as yet the sensations the robot experiences have no meaning for it, and cannot be synthesised to stimulate the robot to original, non-programmed behaviour. Robbie the Robot and his other fictional counterparts are still a long way from reality.

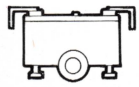
Nevertheless, many products are now being sold under the name 'robot'. These range from small toys for under £1 to vastly expensive R2D2 lookalikes and industrial robots. After examining the components of robot design and theory for several instalments, we must now consider what constitutes a true robot. We must not be too demanding, but we should be able to take what we know and apply it in a workable definition.

The first consideration, and one that eliminates many of the lower-priced 'robot' products, is movement: can the robot move about a space by itself? We cannot expect the robot to program itself, or to set a course of action without human guidance, but we can expect a robot, once set in motion, to be able to operate independently of continuous human control. Without this freedom of movement, an object cannot be considered a robot.

Having passed the test of movement, our robot candidate must now be evaluated on the basis of how the movement is effected. A small toy car can be given a motor and batteries that keep it moving in a straight line. Add bumpers to it, and the car can turn away from obstacles such as walls and tables. Give the car a slightly unusual centre of

Spot The Robot





Our Robot
Powered and controlled from its parent computer, the robot is equipped with touch- and light-sensitive sensors

Big Trak
LOGO-like distance and direction instructions can be programmed into this microprocessor-driven device through its keyboard

Bumper Car
This battery-powered toy will run in a straight line until it hits an object, in which case it will turn clockwise 90° and continue

STEVE CROSS

Amazing Tracks
The three devices are attempting to run a maze: the toy car simply blunders from wall to wall, Big Trak follows its human operator's programmed instructions for running the maze, while our robot learns the maze through the interaction of its software and sensors. We can be sure that the robot will solve the maze eventually, no matter what happens; Big Trak will follow its program, so may solve the maze if the operator's directions are correct; the toy car could solve only 'right-handed' mazes, and then only by chance.

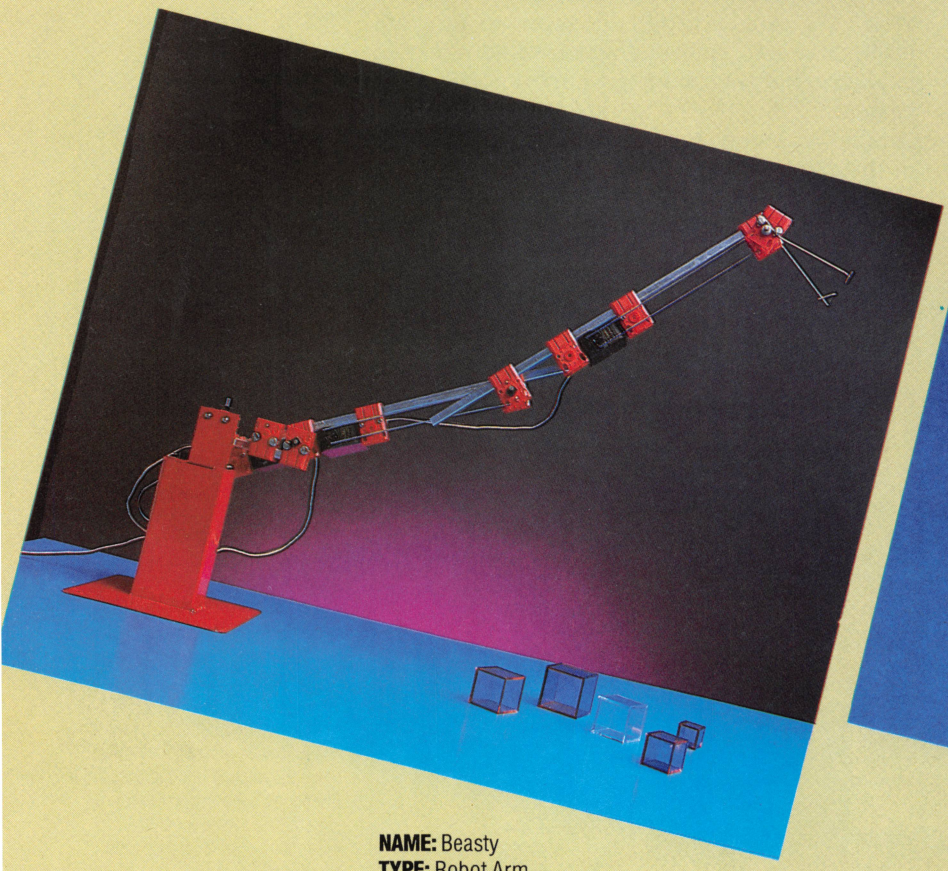
When the car collides with Big Trak, the car is unaffected since its behaviour is purposeless; Big Trak, however, is diverted 90° off its course (shown in green) but continues to turn and travel as if it were still on track (shown in red). Both devices react unintelligently to this unforeseen event where the robot would treat it as just one more aspect of an unpredictable environment

gravity and big rubber tyres and it can even climb up a wall and turn itself over, then continue driving. The car is moving on its own, and is independent of human control, but can it truly be called a robot? The answer, fairly clearly, is 'No', but it is crucial to our study of robots to understand why this is so.

Robot movement, as we have seen, can fall into two categories: simple movement under program control, and intelligent movement. The key to both is programming. Our motor-driven car cannot be a robot because it cannot be programmed to move in different ways. It has a set pattern of operations built into it mechanically, but it cannot respond to human commands and it has no means by which a human controller could generate alternative motion. An interesting variant on this point is the motor-driven car, lorry, or other object that can be programmed by punching a pattern onto a card. The card is read mechanically and the car follows the pattern, turning left or right or continuing straight ahead in accordance with the instructions on the card. By our definition, this type of car would be a robot because it can be programmed with the punched card software. The need for programmable movement eliminates many other small, inexpensive products from the label 'robot'.

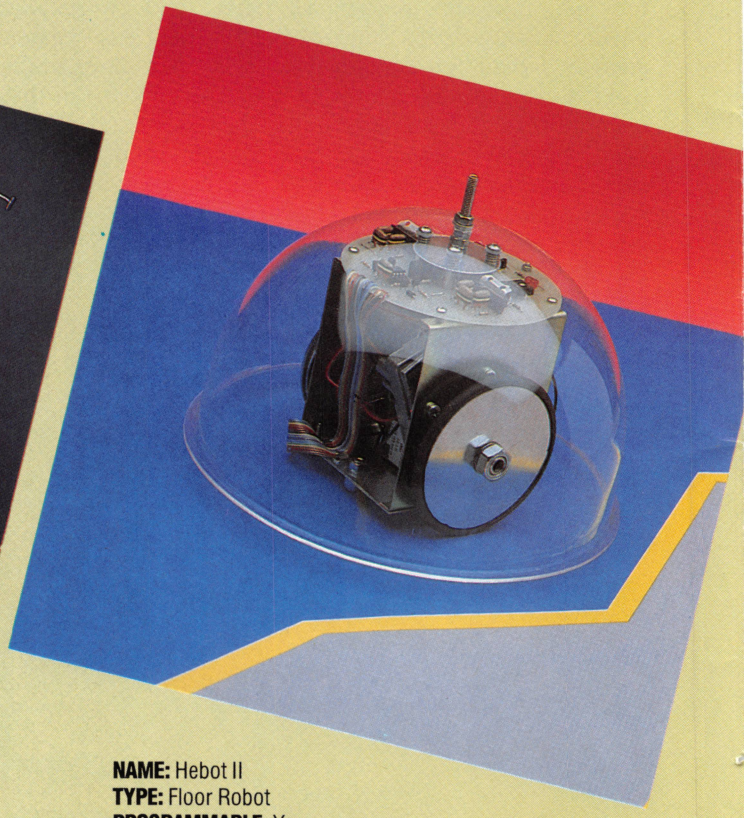
Several other considerations can be applied. Does the object have sensors to give it input from the outside world? Can it respond to its environment and create a changing internal model? Can it play a good game of chess? Any of these criteria can be applied to our robot but, in the final analysis, the element of programmable movement is crucial.

The following products, all billed as robots, are available from stockists for less than £200.



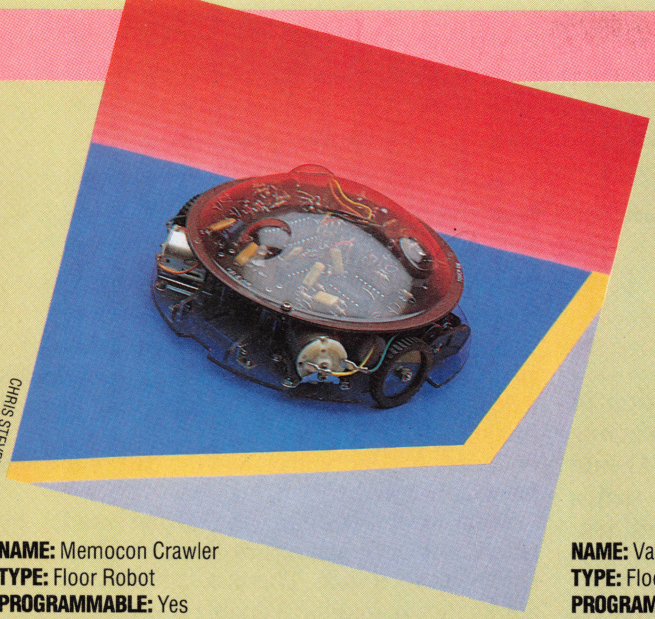
NAME: Beasty
TYPE: Robot Arm
PROGRAMMABLE: Yes
SOFTWARE PROVIDED: Yes
SENSORY FEEDBACK: Positional feedback
PRICE: £110, complete
DISTRIBUTORS: Commotion Ltd, 241 Green Street, Enfield, Middlesex EN3 7SJ

The Beasty receives its commands from the user port of the BBC Micro and is powered from the BBC's auxiliary power supply. It is supplied in kit form, and includes three servo motors. Also supplied are two manuals, a construction booklet and an operating manual. The Beasty comes with its own operating system — Robol — that allows the user to move each of the servo motors independently. (See page 770 for a more detailed review.)



NAME: Hebot II
TYPE: Floor Robot
PROGRAMMABLE: Yes
SOFTWARE PROVIDED: Yes
SENSORY FEEDBACK: Tactile
PRICE: £95 (kit), £169 (assembled)
DISTRIBUTORS: Powertran Cybernetics Ltd, West Portway Industrial Estate, Andover, Hampshire, SP10 3NN

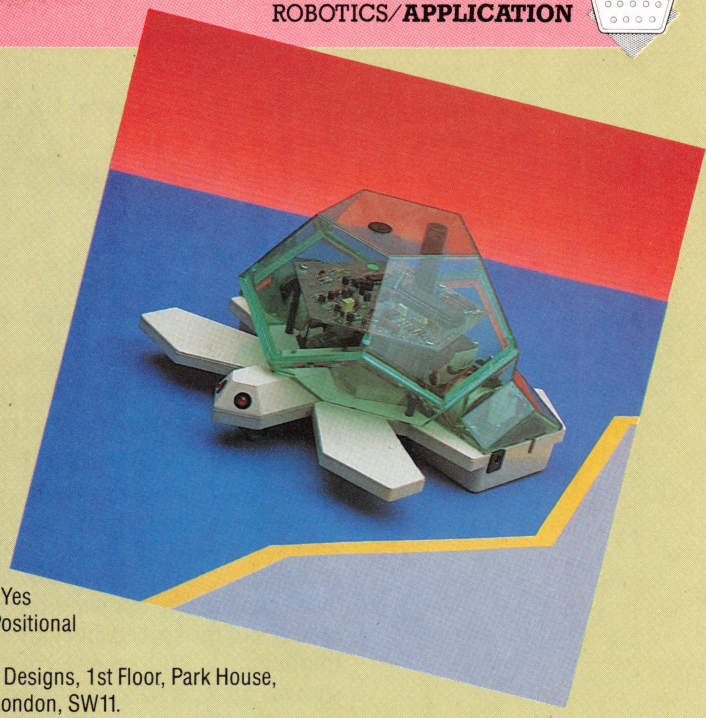
The Hebot II is a robot turtle that is driven by DC motors connected to two wheels. The turtle is interfaced to the edge connector on the Sinclair ZX81, although the manufacturers claim that with some rewiring the Hebot can be made to run from any home micro. The Hebot II comes in kit form with construction booklet. The turtle has a retractable pen and four collision detectors to provide tactile feedback. Power is provided directly from the computer.



CHRIS STEVENS

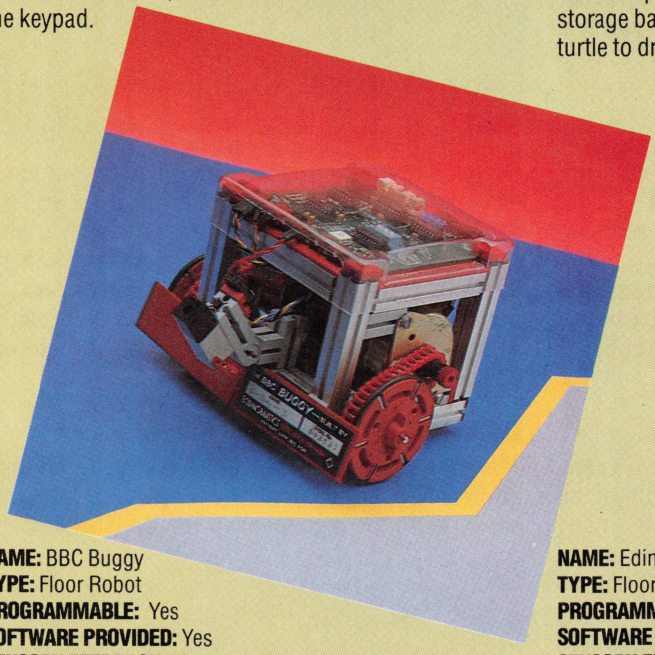
NAME: Memocon Crawler
TYPE: Floor Robot
PROGRAMMABLE: Yes
SOFTWARE PROVIDED: No
SENSORY FEEDBACK: No
PRICE: £34.99
DISTRIBUTORS: Prism Products, Prism House, 18-29 Mora Street, London, EC1.

The Memocon Crawler is easily the most sophisticated robot of the Prism Movits range. The machine uses 1.5v batteries to power two DC electric motors. Control is effected by a box with five keys on it — one for each of the commands available. This is connected to the Crawler via a ribbon cable. The device also has a buzzer and LEDs on board; these can also be controlled from the keypad.



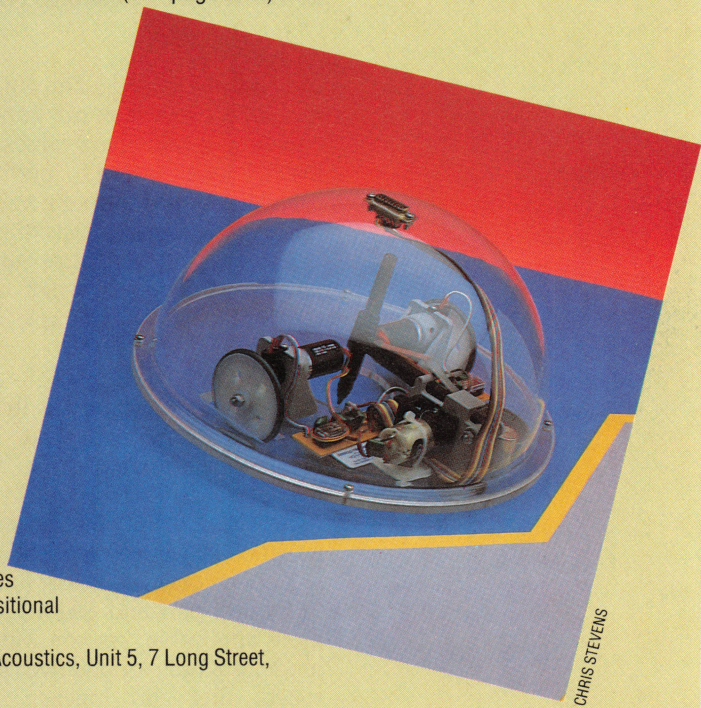
NAME: Valiant Turtle
TYPE: Floor Robot
PROGRAMMABLE: Yes
SOFTWARE PROVIDED: Yes
SENSORY FEEDBACK: Positional
PRICE: £229
DISTRIBUTORS: Valiant Designs, 1st Floor, Park House, Battersea Park Road, London, SW11.

The Valiant Turtle is actually shaped like a turtle. It is driven by a pair of motors, each one providing power for a single wheel. The device is controlled from the computer via an infrared beam. The software provided with the device is designed to be run under LOGO although it will work without the command language. Power is provided from a rechargeable on-board storage battery. There is also a penholder, allowing the turtle to draw designs as it moves. (See page 572.)



NAME: BBC Buggy
TYPE: Floor Robot
PROGRAMMABLE: Yes
SOFTWARE PROVIDED: Yes
SENSORY FEEDBACK: Light sensor, pressure pads
PRICE: £164.35
DISTRIBUTORS: Economatics Education Ltd, Epic House, 9 Orgreave Road, Handsworth, Sheffield

The BBC Buggy is a software-controlled turtle that will already be familiar to many schoolchildren. The Buggy comes in kit form and is run from the user port of the BBC Micro, with power coming from the computer's auxiliary power supply. The Buggy uses a pair of stepper motors that give very precise movement. The BBC Buggy can respond to and search out a light source, and can also be fitted with a pen and a bar code reader.



CHRIS STEVENS

NAME: Edinburgh Turtle
TYPE: Floor Robot
PROGRAMMABLE: Yes
SOFTWARE PROVIDED: Yes
SENSORY FEEDBACK: Positional
PRICE: £192
DISTRIBUTORS: Jessop Acoustics, Unit 5, 7 Long Street, London, E2.

The Edinburgh Turtle is named after the city in which it was developed. The turtle is connected to the computer via a ribbon cable, from where it also draws its power. It is driven by a pair of DC electric motors, each of which powers a single wheel, and is fitted with a retractable penholder and an on-board speaker. Jessop has recently introduced a remote-controlled version of this turtle that retails at just over £200. (See page 572.)

EDUCATED GUESS

We conclude our two-part examination of TK!Solver — an equation processing program for the Apple II, IBM PC and compatibles, and the ACT Apricot — with a closer look at some of its unique abilities.

As we explained in the previous instalment of our spreadsheet series (see page 804), TK!Solver is a 'next generation' software package that takes the concept of the spreadsheet into the realm of higher mathematics and engineering. We have already shown that TK!Solver lets the user define variables with names and use these in complex mathematical equations. In this instalment, the last of our spreadsheet series, we look in detail at TK!'s unusual ability to *iterate*. This is a method where the program can solve for a variable by guessing at it. Ordinarily, when working with equations, one can determine the values of all the variables if enough information is given from the outset. The program simply reduces the problem to a series of calculations. For example:

$$A^2 + B^2 = 2COS\ Y$$

can easily be solved for any of the three variables if the other two values are known. Faced with this equation — and given values for A and B — TK!s Direct Solver would perform the required calculations and output a value for Y.

But there are occasions when the determination of a value is not straightforward. One such case is a redundant equation, which defines a variable in terms of itself. For example, consider:

$$D=(A+B)/(2*D)$$

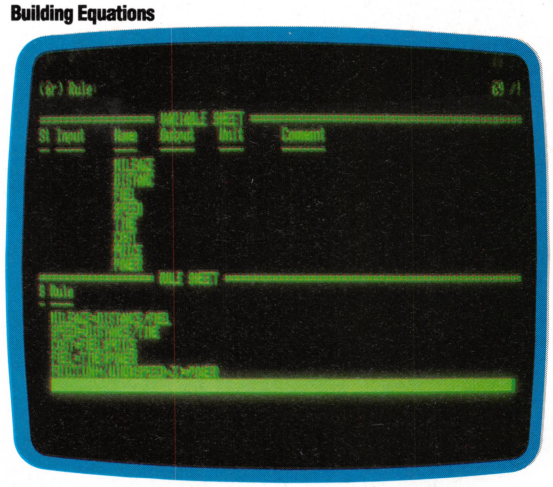
within a model where A is the only known value. Other problems can occur as the result of an incomplete model, or a model with many interdependent variables and a limited amount of data. The concept of iteration is a difficult one, so let's look at a more practical example of iteration.

Let's reconsider the car journey model created in the last part and add a few details to make it more applicable. As you will recall, the previous model was built around five values: distance, time, speed, fuel and mileage. It could calculate mileage, given speed and fuel consumption; distance from speed and time; and several other simple variations. What if we now want to determine how fast we should travel in order to complete a trip within a given budget?

To begin with, we must add several factors to our model. For instance, the model must take into account the power output of the vehicle, the internal friction of the engine and wind resistance, all of which will have an effect on the vehicle's mileage and speed. (We will assume that internal friction is constant.) We must also have an upper boundary for our budget, and the cost of the fuel being consumed.

We'll begin building the actual model by entering these equations in the Rule sheet, one equation to a line. These equations are read automatically into the Variable sheet:

Building Equations



Because there are several variables, the screen is too small to hold all the information. To see all the variables displayed, we can show the Variable sheet in a window by itself. We do this by pressing the semi-colon key (;) to move the cursor into the variable window, and then type in W1. Now all the variables can be seen and we can begin entering values for them.

DIRECT SOLVING

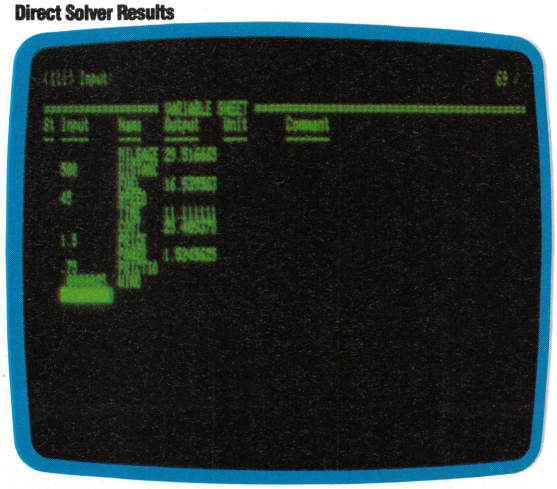
The model can be solved directly, if enough information is given at the start. For instance, enter the following values in the INPUT column:

Input Values For Direct Solver

IAN McKINNELL

Then press ! to perform the calculation. TK! displays the message Direct Solver, and the unknown values appear in the OUTPUT column, as shown:

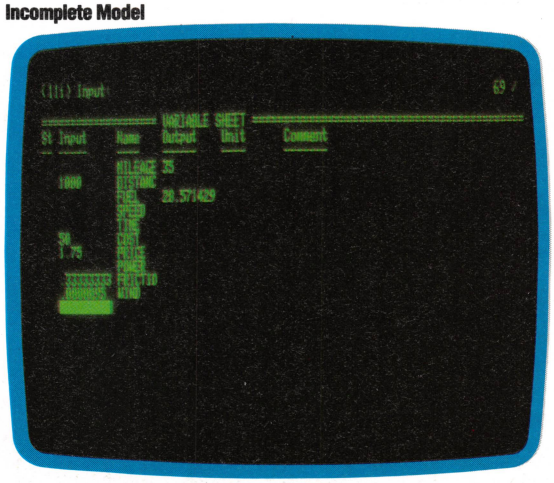
Direct Solver Results



This gives us a nice breakdown of all the information we want. But what if we want to use this model to solve a problem with less information given at the outset? Let's consider a calculation where we have a maximum budget of £50 to spend on fuel for a 1,000 mile trip. We know the price of fuel (say £1.75 per gallon) so we can easily determine how much we can spend per mile. It might be more difficult, however, to determine what speed we need to travel in order to achieve the mileage needed to complete the trip within our budget.

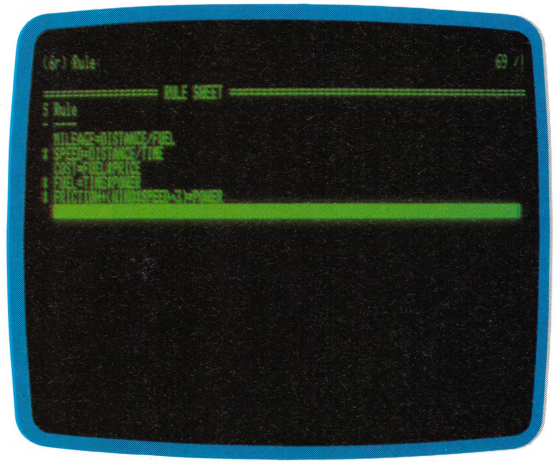
We begin by blanking the values already entered. We do this by typing RVY (for Reset Variables Yes). Then we type in the information that we know: 1000 for distance, 50 for cost, and 1.75 for price. We use a value of 1/3 for internal friction (which TK! evaluates to 0.333333) and 0.0000095 for wind resistance. Press ! to calculate and the following values appear:

Incomplete Model



Note that no values have been generated for speed, time or power — and speed is the specific value we need. If we shift the display from the Variable sheet to the Rule sheet, we will see that three of our equations remain unsatisfied (which is indicated by the * in the Status column):

Unsatisfied Equations

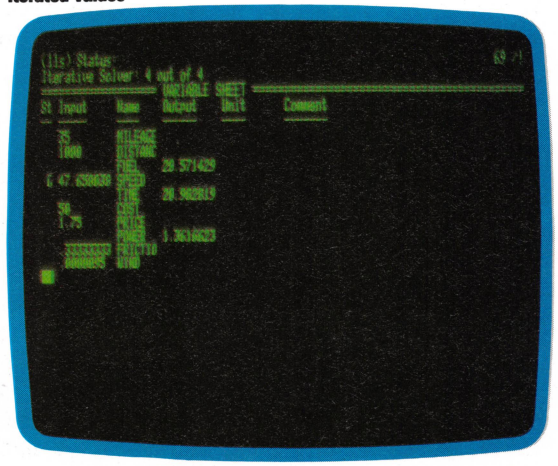


ITERATIVE SOLVER

Since we cannot solve the model using the Direct Solver, we must try the Iterative Solver. This takes a starting value, input as a guess, and fits it into the equation. If the value is not correct, TK!Solver uses a series of successive approximations (like the game of 'higher' and 'lower') to pinpoint the exact value.

The first thing we do is take the mileage value generated previously and move it into the Input column to give TK! one extra value to start with. We do this by typing I in the Status column next to mileage on the Variable sheet. Then we estimate a value for speed — say 50 — enter that number in the Input column, type G for guess in the Status column and press ! to calculate. TK! displays Iterative Solver at the top of the screen and counts off each approximation. On the fourth attempt, TK! arrives at the correct value for speed, time and power, as shown:

Iterated Values



According to TK!Solver, an average speed of just over 47 miles per hour is needed to complete the trip within our budget. The closer a guess is to the actual value, the sooner TK! finds a solution.

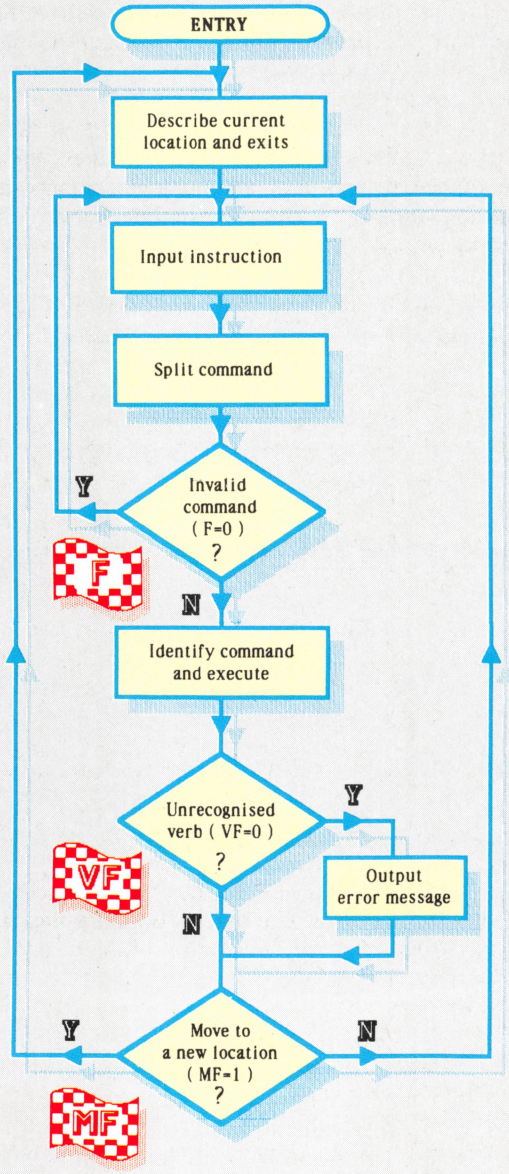
TK!Solver is available from Practicorp for the Apple II, IBM PC and compatible machines, and the ACT Apricot, for £195. Software Arts also publishes 'Solver Packs' at £95 each with predesigned models for specific applications.

TAKING ORDERS

Up to this point in our adventure game programming project, we have discussed methods of map making, formatting output and moving around the adventure world. In this instalment, we show how the program analyses and obeys instructions given to it by the player.

Chequered Flags

Flags are used widely in programs that have a modular construction. Conditions that involve branches in program control can be tested for within a module but branching on the result of such a test can be delayed until a return to the main program section is made. By setting a variable to a pre-determined value when the test is made, the value of the variable can later be tested within the main program section. Variables used to indicate conditions in this way are termed 'flags'. The flowchart shows the main program loop, as constructed so far, for Haunted Forest. The flag F indicates whether or not a command has a valid format and is set during the 'split command' subroutine. The subroutine used to identify and execute normal commands uses two flags: VF is used to signal that the verb part of the command has been correctly recognised. If, in executing a command, the player moves to a new location, the fact that a move is to be made is signalled by MF. When MF is tested within the main program loop, a value of 1 indicates that the loop should branch back to describe the new location to which the player has moved



IAN McKINNELL

Adventures are usually constructed so that the player can move from location to location, picking up and dropping objects along the way. A set of commands allows the player to perform these simple tasks. The commands we have used are:

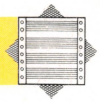
GO (direction)	To move between locations
TAKE (object)	To pick up an object
DROP (object)	To put down an object
LIST	To list the objects carried
LOOK	To redisplay the description of the current location
END	To end the game

Variations on these may also be available, such as MOVE instead of GO, or GET instead of TAKE. Part of the fun of playing an adventure game is to determine what words the game will accept. For example, a player might try the command SWIM when in a dry location. If the program responds by telling the player that he cannot swim *here*, then the player could reasonably assume that there are locations where swimming *is* allowed. (Alternatively, the programmer might just want the player to think that!)

The number of commands accepted by a game varies according to the complexity of the game and the amount of effort the programmer has made to cover every eventuality. The most important thing for the designer to do is to make sure the program does not crash if a player tries to enter a command that is not catered for. A failsafe routine that prints 'I don't understand' may be all that's required, bearing in mind that some flexibility should be added so that players can enter commands in different ways. For example, it would be annoying for a program that accepts the command TAKE LAMP to respond to the command TAKE THE LAMP with 'I don't understand'. Adding flexibility will be discussed at greater length later. For the moment, we need to look at the type of instructions that might be given during the game, and devise a routine that will break these down into a form that can be easily interpreted.

COMMAND SPLITTING

No matter what the instruction is, it is very likely that it will be phrased in the *imperative* — such as, GO SOUTH TOWARDS THE RIVER or KILL THE ALIEN. The advantage of this sentence structure is that it is easy to break down: the verb always comes as the first word in the sentence, the object of the verb follows this, and finally there may be some form of qualification of the action. A first stage in the analysis of a command is to separate the verb from



the rest of the sentence. This task can be easily achieved by scanning through the sentence one character at a time, using MID$, until a space is found. The part of the sentence that lies to the left of the space is the verb, and can be assigned to the variable VB$. The part of the sentence to the right can be assigned to a second variable, NN$. This subroutine is used in Haunted Forest to split the instruction assigned to the variable IS$:

```
2500 REM **** SPLIT COMMAND S/R ****
2510 IF IS$="LIST" OR IS$="END" THEN VB$=IS$:F=1:RETURN
2515 IF IS$="LOOK" THEN VB$=IS$:F=1:RETURN
2520 F=0
2530 LS=LEN(IS$)
2540 FOR C=1 TO LS
2550 A$=MID$(IS$,C,1)
2560 IF A$<>" " THEN 2590
2570 VB$=LEFT$(IS$,C-1):F=1
2580 NN$=RIGHT$(IS$,LS-C):C=LS
2590 NEXT C
2600 :
2610 IF F=1 THEN RETURN
2620 PRINT:PRINT"I NEED AT LEAST TWO WORDS"
2630 RETURN
```

Before the routine attempts to split up the sentence, it first checks to make sure that the command is not one of the three possible single-word instructions — that is, LIST, LOOK or END. If it is a single-word command, then the complete instruction is assigned to VB$, and the routine is exited. If the command is not one of these, then the routine enters a FOR...NEXT loop and begins to scan for the first space. Two techniques used within this loop need special mention. Both relate to the fact that it is extremely bad programming style to perform a conditional jump out of a FOR...NEXT loop without passing through the NEXT statement. Instead, to signal the fact that some condition has been met — in this particular case, that a space has been found — a flag, F, is set to one. Secondly, when the first space has been found, it is a waste of time to continue scanning through the rest of the sentence.

The loop can be neatly terminated at this point by setting the loop counter, C, to its upper limit, LC. Consequently, when the program again reaches NEXT, it will pass on to the following instruction, rather than loop back to the FOR statement. Once the loop has been correctly terminated, then the status of the flag, F, can be tested. A flag value of one indicates that the sentence consists of more than one word, and all that remains to do at this stage is to return to the main loop. If the flag is not one, then the command has only one word and is not one of the single-word commands tested for earlier. In this case, a message stating that two words are required is printed before returning for another command.

NORMAL COMMANDS

For the main part of the program, the player will simply move from location to location and pick up or drop objects that may be found. Therefore, for the majority of locations, the commands GO, TAKE, DROP, LIST, LOOK, END — and their variants — are sufficient to allow the player to do this. Only in unusual circumstances will the player wish to use other more specialised commands. For example, there is little point in using the command KILL if there is nothing present to kill. We can, however, devise a program structure where, on the majority of occasions, only the six commands associated with movement and objects are tested for. When the player enters a new location, the program can test to see if it is one that has been designated 'special' in some way. If this is the case, then any new command requirements can be dealt with by a specific command subroutine for that particular location. Therefore, the main calling loop to our program should do the following:

1) Describe the location and list the exits.
2) Determine whether the location is 'special'.
3) Ask for a command and, if the location is not special, scan the list of normal commands.

There must also be a facility in the main loop to distinguish between a command that causes a move to a new location and one that does not. In the first case, the loop needs to go back to the beginning of the loop to describe the new location and decide whether or not it is special. In the second case, it is necessary only to loop back to ask for a new command. The simplest way to implement this is to use a 'move flag', MF, which is normally set to zero. If a command involves movement then this flag is set to one. The status of MF can be tested at the end of the main loop and the appropriate jump made. Add the following lines to Haunted Forest:

```
270 GOSUB2500:REM SPLIT INSTRUCTION
275 IF F=0 THEN 260:REM INVALID INSTRUCTION.
280 GOSUB3000:REM NORMAL COMMANDS
290 IF VF=0 THENPRINT:PRINT"I DONT UNDERSTAND"
300 IF MF=1 THEN 240:REM NEW LOCATION
310 IF MF=0 THEN 260:REM NEW INSTRUCTION

3000 REM **** NORMAL COMMANDS S/R ****
3010 VF=0:REM VERB FLAG
3020 IF VB$="GO" OR VB$="MOVE" THENVF=1:GOSUB3500
3030 IF VB$="TAKE" OR VB$="PICK"THEN VF=1:GOSUB3700
3040 IF VB$="DROP" OR VB$="PUT"THEN VF=1:GOSUB3900
3050 IF VB$="LIST" OR VB$="INVENTORY"THEN VF=1:GOS
     UB4100
3055 IF VB$="LOOK" THEN VF=1:MF=1:RETURN
3060 IF VB$="END" OR VB$="FINISH" THEN VF=1:GOSUB4
     170
3070 RETURN
```

In the first routine, another flag, VF, is used to indicate whether or not the verb has been understood and obeyed. Only when the verb has been isolated is VF set to one. We can insert a failsafe 'I don't understand' statement in the main loop by testing the status of VF. If VF remains zero then the verb in the command has not been recognised by the analysis routine, and the statement is displayed.

In the next instalment of the project, we will deal with subroutines for picking up, dropping and listing objects. For now, we can add a short END command subroutine to our group of normal commands:

```
4170 REM **** END GAME S/R ****
4180 PRINT:PRINT"ARE YOU SURE (Y/N) ?"
4190 GET A$:IF A$<>"Y" AND A$<>"N" THEN 4190
4200 IF A$="N" THEN RETURN
4210 END
```

The LOOK command is also straightforward. To redescribe the current position, we simply need to set the 'move flag', MF, to one and return to the main program loop. Setting MF will cause the main

program to loop back to the beginning, thus calling the routines that describe a location and its exits. As the value of the location variable, P, is not changed by the LOOK command, the same location will be described. This command is useful if, after the player has performed a series of actions, the original description of the current location has moved off the screen.

ADDING FLEXIBILITY

When issuing movement commands, the player may type in different forms of the same instruction. For example, GO NORTH, MOVE NORTH and GO TOWARDS THE NORTH are asking the same thing. Although it is not vital for an adventure game program to recognise all of these forms, it makes playing the game more interesting if a number of different instruction formats are legal. The three movement commands we just gave have a common structure: they all start with a movement verb, and the direction required is a discrete word. It is possible, therefore, to design a routine that will search the part of the sentence coming after the verb for the direction. The routine scans this part of the sentence for spaces, isolating each word in turn and comparing it with the four direction words sought until a match is found.

```
3630 REM **** SEARCH FOR DIRECTION S/R ****
3640 NN$=NN$+" ":LN=LEN(NN$):C=1
3645 FOR I=1 TO LN
3650 IF MID$(NN$,I,1)<>" " THEN NEXT I:RETURN
3655 W$=MID$(NN$,C,I-C):C=I+1
3660 IF W$="NORTH" OR W$="EAST" THEN NN$=W$:I=LN
3665 IF W$="SOUTH" OR W$="WEST" THEN NN$=W$:I=LN
3670 NEXT I
3675 RETURN
```

In the last instalment of the project we developed a movement routine. To add this new routine to the movement routine we need simply to add the following line:

```
3505 GOSUB3630:REM SEARCH FOR DIRECTION
```

It is worth noting that this routine will not obey instructions such as GO IN A NORTHERLY DIRECTION, since the direction word cannot be isolated by the routine. It would be possible to design a routine that worked on the principle of scanning groups of four and five letters, comparing each group with the four possible direction words. However, such a routine would have a long execution time. On the other hand, our program will accept GO NORTHWARDS, as the movement routine finally uses the first letter of the second part of the sentence, NN$. In this case, the N in NORTHWARDS would be accepted as N for NORTH.

Digitaya Listings

```
1220 GOSUB1700:REM ANALYSE INSTRUCTIONS
1225 IF F=0 THEN 1210:REM INVALID INSTRUCTION
1230 GOSUB 1900:REM NORMAL INSTRUCTIONS
1240 IF VF=0 THENPRINT"I DON'T UNDERSTAND"
1250 IF MF=1 THEN 1160:REM NEW POSITION
1260 IF MF=0 THEN 1210:REM NEW INSTRUCTION

1700 REM **** ANALYSE INSTRUCTION S/R ****
1705 F=0:REM ZERO FLAG
1710 IFIS$="END" OR IS$="LIST" THEN VB$=IS$:F=1:
     RETURN
1720 IF IS$="LOOK" THEN VB$=IS$:F=1:RETURN
1730 :
1740 REM ** SPLIT INSTRUCTION **
1750 VB$="":NN$="":REM ZERO VERB AND NOUN
1770 LS=LEN(IS$)
1780 FOR C=1TO LS
1790 A$=MID$(IS$,C,1)
1800 IF A$=" "THEN VB$=LEFT$(IS$,C-1):NN$=RIGHT$(I
     S$,LS-C):F=1:C=LS
1810 NEXT
1830 IF F=0 THEN PRINT:PRINT"I NEED AT LEAST TWO
     WORDS"
1840 RETURN
1850 :
1900 REM **** NORMAL ACTIONS S/R ****
1910 VF=0
1920 PRINT
1930 IF VB$="GO"ORVB$="MOVE"THENVF=1:GOSUB2000
1940 IF VB$="TAKE"ORVB$="PICK"THEN VF=1:GOSUB2140
1950 IF VB$="DROP"ORVB$="PUT"THENVF=1:GOSUB2360
1960 IF VB$="LIST"ORVB$="INVENTORY"THENVF=1:
     GOSUB2540
1965 IF VB$="LOOK" THEN VF=1:MF=1:RETURN
1970 IF VB$="END"ORVB$="FINISH"THENVF=1:GOSUB2610
1980 RETURN

2015 GOSUB8600:REM SEARCH FOR DIRECTION

2610 REM **** END GAME S/R ****
2620 PRINT:PRINT"ARE YOU SURE (Y/N) ?"
2630 GETA$:IFA$<>"Y"AND A$<>"N"THEN2630
2640 IFA$="N"THEN RETURN
2650 END

8600 REM **** SEARCH FOR DIRECTION S/R ****
8610 NN$=NN$+" ":LN=LEN(NN$):C=1
8620 FORI=1 TO LN
8630 IF MID$(NN$,I,1)<>" " THEN NEXT I:RETURN
8640 W$=MID$(NN$,C,I-C):C=I+1
8650 IF W$="NORTH" OR W$="EAST" THEN NN$=W$:I=LN
8660 IF W$="SOUTH" OR W$="WEST" THEN NN$=W$:I=LN
8670 NEXT I
8680 RETURN
```

Basic Flavours

Spectrum:

In both programs, use I$ for IS$, B$ for VB$, and R$ for NN$ throughout.

Replace the following lines in Digitaya:

```
1790 LET A$=I$(C TO C)
1800 IF A$=" " THEN LET B$=I$(TO C-1):LET R$=I$
     (LEN(I$)-LS+C+1 TO):LET F=1:LET C=LS
2630 LET A$=INKEY$:IF A$<>"Y"
     AND A$<>"N" THEN 2630
8630 IF R$(I TO I)<>" " THEN NEXT I:RETURN
8640 LET W$=R$(C TO I-1):LET C=I+1
```

Replace these lines in Haunted Forest:

```
2550 LET A$=I$(C TO C)
2570 LET B$=I$(TO C-1):LET F=1
2580 LET R$=I$(LEN(I$)-LS+C+1 TO):LET C=LS
3650 IF R$(I TO I)<>" " THEN NEXT I:RETURN
3655 LET W$=R$(C TO I-1):LET C=I+1
4190 LET A$=INKEY$:IF A$<>"Y" AND
     A$<>"N" THEN GOTO 4190
```

BBC Micro:

Replace this line in Digitaya:

```
2630 REPEAT:A$=GET$:UNTIL A$="Y" OR
     A$="N"
```

and this line in Haunted Forest:

```
4190 REPEAT:A$=GET$:UNTIL A$="Y" OR
     A$="N"
```

INFORMATION STORAGE AND RETRIEVAL

Information storage and retrieval is the term used for the accessing and retrieval of information from a magnetic medium such as disk, tape or — in older systems — cards or punched tape. This information may be in the form of files, programs, data or graphics. It is necessary to use a storage medium that will not only retain the information when the power is switched off, but also allow the computer to read the information back in at some other time. This means that a computer must have, or be attached to, an interface that can read and write the data. Thus information storage and retrieval has come to mean not only the process of reading and writing information, but also the techniques involved in such a process.

INFORMATION TECHNOLOGY

In one sense, *information technology* has been around for thousands of years: hieroglyphics, the abacus, the quill pen and the printing press are all examples. In the last decade or so, following the development of microelectronics and the subsequent dramatic fall in the price of such circuitry, information technology (IT) has come to mean the electronic storage, transmission and processing of information — specifically with reference to computers, video and telecommunications. As such, IT has already had a profound effect on much of the world's population, and most major organisations are now dependent on computers. An enormous amount of effort and resources is currently being put into information technology. The world now depends for its communications on satellites, such as COMSAT, while other satellites constantly monitor every inch of the world's surface.

INFORMATION THEORY

Information theory (developed by Claude Shannon at Bell Laboratories, New Jersey, in 1948) is the area of computing that investigates the transmission of data. It involves the examination of newly-arrived data and whether it tells us something new — that is, the extent to which it reduces the *uncertainty* in the information system. Thus, information theory consists of the study of the nature of the information and its speed of arrival. In its simplest form, this restricts information theory to the rate at which new information arrives, and from which channel or source.

However, in a wider sense, information theory can include such areas as coding theory. This discipline covers the translation of data from one form to another, and how this can be accomplished efficiently without any loss of the information being transmitted.

INITIALISATION

Primarily, *initialisation* is the process performed by the computer's ROM-based operating system when the machine is switched on. This involves default values being placed in the registers and various addresses in memory. Typically, during initialisation, a microcomputer will set the stack pointer, clear the decimal mode and initialise the various input/output devices. This initialisation will also set the top and bottom of memory pointers and the zero page. Finally, initialisation includes the setting up of the initial screen display and the screen editor variables.

Initialisation has a second meaning: the formatting of disks. This takes the form of writing the track that will contain the disk directory. The information written on this track is the disk's title, the block availability map (BAM) and a list of markers for each track on the disk.

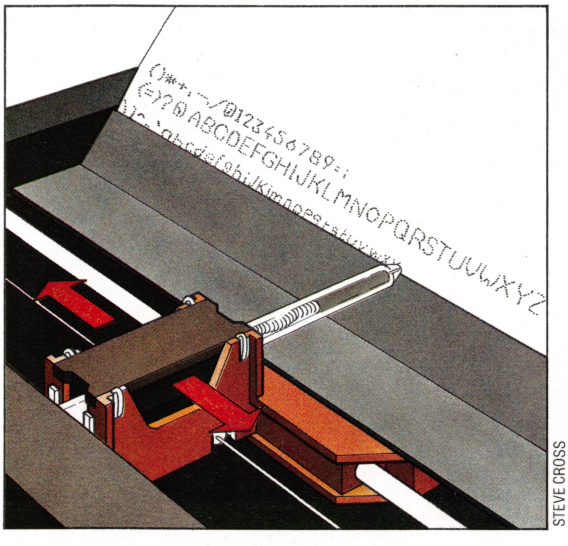
Initialisation can also be performed by a programmer. If a program is to be properly structured, variables should have initial values assigned to them at the beginning of a program. This is often termed the 'initialisation procedure'.

INK JET PRINTER

An *ink jet printer* forms characters by squirting droplets of ink from a nozzle at the piece of paper. Although these printers were developed in the 1960s, it was not until the mid-1970s that their use became widespread.

There are two types of ink jet printer. The *pulsed* jet printer consists of either one or a number of nozzles, which fire several droplets to produce a single dot on the paper. By arranging these dots in patterns, characters can be built up — in much the same way as a dot matrix printer forms its type. If a number of nozzles are fitted to the print head, then the user has the possibilty of multicolour or higher resolution printing.

The other type of ink jet printer is known as the *continuous stream* printer. As its name suggests, this printer shoots a continuous jet of ink at the paper. The droplets in the stream are electrically charged as they leave the nozzle, and then pass between charged electrodes, which alter the direction of the ink stream. The droplets of ink are directed to a precise position, and the character is drawn in much the same way as a pen stroke.



A Drop In Quality
Ink is fired at the paper through a nozzle that breaks the stream into separate droplets. These are electrically charged, which enables the metal deflector plates to direct them into character patterns on the paper — the electron beam in a cathode ray tube is moved around the screen in exactly the same way

A TOUCH OF CLASS

Many different devices have been marketed as aids in the construction of graphics displays. The Touchmaster graphics tablet is unique in that it may be used with most of today's popular home computers and, so the manufacturer claims, can also be used as a simplified replacement keyboard.

Today's best-selling computers all support high-resolution graphics displays. However, unless ready-written graphics software is available, much time and effort is required to create such displays and many features are not fully utilised. A 'sketch' program is not sufficient because the user will often wish to copy an existing image into the computer instead of simply drawing freehand.

Several digitisers have been marketed for this purpose, but these have mostly been designed for use with specific machines, such as the BBC Micro or ZX Spectrum. The Touchmaster graphics tablet is designed to work with a wide range of home machines (some of which will require a suitable interface or cable). This device is also being promoted as a replacement keyboard, but the simplicity of its design means that such use is restricted to selection between a number of menu options or for simple games control. A computer keyboard is still required for data entry, as well as for loading the Touchmaster software itself.

The Touchmaster is fitted in a neat grey case measuring 350 by 330 by 35mm. The back of this is slightly raised, forming a convenient angle for drawing. A plug-in transformer is supplied, with a single red LED indicating when power is on; however, no on/off switch is fitted. To allow the tablet to be used with a wide range of home machines, both serial and parallel interface sockets are fitted to the rear panel, together with a socket — not mentioned in the manuals — for a foot switch. In fact, the manuals are barely adequate: the hardware manual gives instructions on the connection of the tablet and provides a number of simple BASIC programs for reading co-ordinates, but is insufficiently detailed.

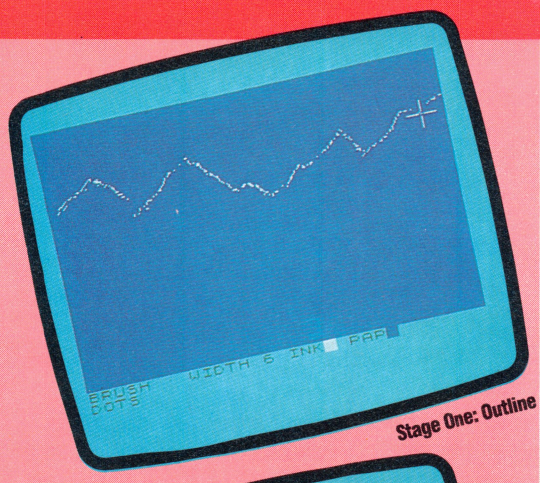
The tablet relies on the membrane technology that was developed on the ZX81 and Spectrum keyboards, and provides a 256 by 256 pixel resolution. The upper layer is separated from the lower resistive film by an insulating mesh, and pressure on the upper layer forces it to make contact with the film. The tablet contains a microprocessor that scans the top film in one direction while scanning the lower layer in another, and the co-ordinate of the 'contact point' is then sent over both serial and parallel interfaces. The serial interface is used to connect the tablet to the BBC Micro, while the parallel interface is required for use with the Commodore 64, Vic-20, Spectrum and Dragon. The Touchmaster's resolution is less than that provided by many hi-res screen displays, so BBC Micro owners, for example, will be unable to resolve to a single pixel in Mode 0.

IAN McKINNELL

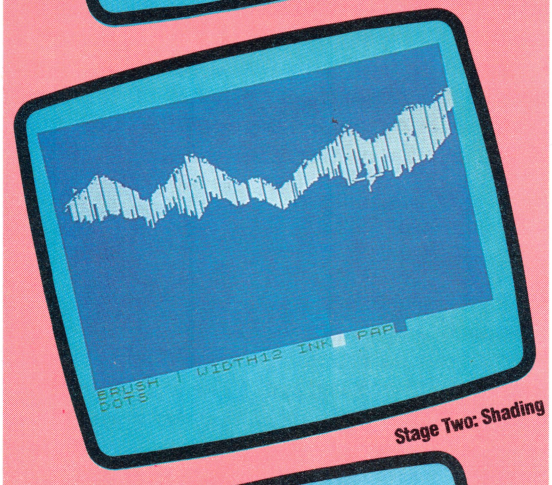
The Touchmaster Pad
The sheet provided with the software simply slots into the drawing area. By pressing the appropriate command on the right-hand side of the overlay and moving the pencil, stylus or finger to the drawing area the command will be executed on the screen

MULTIPAINT PROGRAM

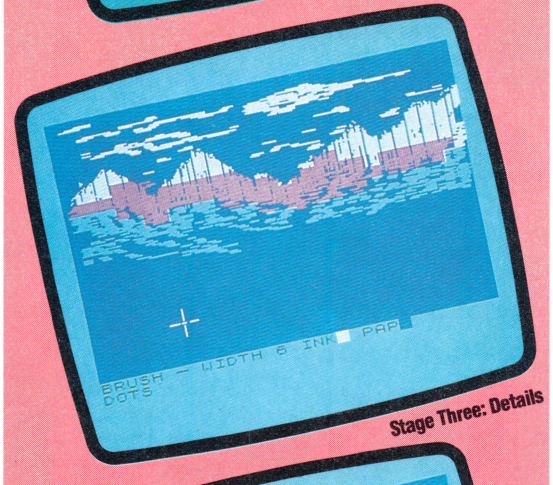
A drawing program called Multipaint is supplied with the Touchmaster. This provides a demonstration of the facilities provided, but is hardly a comprehensive graphics aid. A plastic template gives a menu of the facilities available, with the selected option displayed in a 'status' window at the bottom of the screen. Five different brush types may be used; each of these can be any width from two to 32 pixels, in steps of two pixels. The window also shows the current drawing mode — Dots, Points or Freehand — and the selected



Stage One: Outline



Stage Two: Shading



Stage Three: Details

Stage Four: Colour

A Touching Scene

Touchmaster can be used as a graphics tablet with the Multipaint software and overlay that are supplied with the product. We show stages in the development of a scene from line sketch to coloured and shaded image

foreground and background colours. Colours may be changed by pressing the required option on the menu until the desired choice appears in the status window.

Once the correct colours and brush types have been selected, further options are avilable for the creation of boxes, circles, polygons and 'rubber-banded' lines. A stylus is supplied with the package, but finger pressure may be used instead. The Touchmaster, with its larger tablet area, does not suffer from the same restrictions as the Koala-pad (see page 629): pressure from a finger may be translated into a precise co-ordinate rather than a mere approximation, so electronic finger-painting is a real possibility!

Unfortunately, Multipaint offers no more than rudimentary features. A FILL option is marked on the template and documented in the manuals, but — on the Spectrum version at least — this does not appear to operate in the expected manner. Neither is there any facility for magnification or editing, which means that colours may not be changed. On the Spectrum, where it is often easier to draw in black and white before adding colour, this is a decided disadvantage.

As a piece of hardware the Touchmaster tablet appears to have a lot going for it compared with rivals like the Grafpad and the Koala-pad. It is robustly built and offers a full A4-sized drawing area that can be connected to most of the more popular home computers. One significant advantage is that if you decide to upgrade or change your machine in the future, all that is required is a new interface — plus the appropriate software, of course.

It is a disappointment that the documentation and software supplied should be so poor compared with the standard of the tablet itself. Touchmaster is bringing out a range of software designed specifically for use with this tablet; although the real proof of success will come if independent software houses decide to support it.

TOUCHMASTER

PRICE
£149.95, additional software packages £9.99

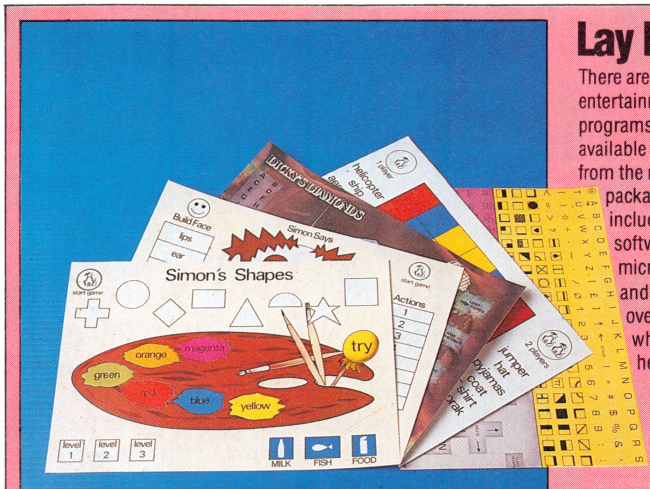
DIMENSIONS
350×330×35 mm

INTERFACES
Serial and parallel ports are provided for use with a wide range of home computers or the Touchmaster's own keypad

DOCUMENTATION
Although the manuals contain the necessary information to use the Touchmaster, more detailed information would have been welcome

STRENGTHS
The wide range of interfaces, together with the different software packages, make the Touchmaster an extremely adaptable peripheral

WEAKNESSES
The quality of the software compares badly with similar devices and the manuals are limited



Lay It On Me

There are various games, entertainments, educational programs and utilities available for Touchmaster from the major retailers. Each package costs £9.99 and includes the appropriate software (for a range of micros), instructions and the imaginative overlays — some of which we illustrate here

WHO DUNNIT?

Our series of investigations into the use of LOGO's list processing facilities continues with a look at how to set up a simple database. We use the example of a murder investigation, in which a list of suspects is created and then analysed to ascertain who the murderer was.

A terrible murder has been committed in a small community in the Ozark Mountains. Zachariah has been viciously attacked with an axe and killed. We know that Matthew and Joshua both own axes, James and Ebenezer own guns, and cousin Jane has a knife. Matthew and James both had blood on their hands when they were questioned by the local sheriff.

Our LOGO database of information about this crime will consist of a list of *facts* — each of which consists of a *relation,* together with one or more nouns. When represented in LOGO, one fact is [OWNS MATTHEW AXE] or, in English, 'Matthew owns an axe'. To represent the fact that James had blood on his hands, we use [BLOODY JAMES].

We begin our investigation with an empty database:

```
TO SETUP
   MAKE "DATABASE []
END
```

We then add facts to our database as we discover them (providing they are not already in the database). For example, we would input ADD [OWNS JANE KNIFE] using the following ADD procedure:

```
TO ADD :FACT
   IF NOT MEMBER? :FACT :DATABASE THEN
      MAKE "DATABASE FPUT :FACT :DATABASE
END
```

The database will eventually fill up:

```
[[BLOODY MATTHEW][BLOODY JAMES][KILLED
   ZACHARIAH AXE] [OWNS MATTHEW AXE]
   [OWNS JOSHUA AXE] [OWNS JAMES GUN]
   [OWNS EBENEZER GUN] [OWNS JANE KNIFE]]
```

To print out the database use SHOW. This can be followed by either "ALL, in which case the whole database will be printed, or by the name of a relation, in which case only the facts for that relation are printed. So, SHOW "OWNS will show us who owns what.

```
TO SHOW :S
   IF :S = "ALL THEN LIST.ALL :DATABASE
   LIST.REL :S :DATABASE
END
```

```
TO LIST.ALL :LIST
   IF EMPTY? :LIST THEN STOP
   PRINT FIRST :LIST
   LIST.ALL BUTFIRST :LIST
END
TO LIST.REL :S :LIST
   IF EMPTY? :LIST THEN STOP
   IF :S = FIRST FIRST :LIST THEN PRINT FIRST
      :LIST
   LIST.REL :S BUTFIRST :LIST
END
```

Now we must devise ways of querying the database. The simplest kind of query we might make of our database is to check whether a fact is known to be true. This we do with a procedure called DOES, which checks whether a fact is in the database. For example, DOES [OWNS JANE KNIFE] should give the answer YES.

```
TO DOES :FACT
   IF MEMBER? :FACT :DATABASE PRINT "YES
      ELSE PRINT "NO
END
```

It would be much more useful for our investigation into this terrible murder if we could ask questions such as 'Who owns an axe?'. The way we will deal with this is to use 'variables'. Any word whose first character is ? will be assumed to be a variable. We can then paraphrase the question as:

```
WHICH [OWNS ?SOMEONE AXE]
```

The reply to this will be a list of all possible values of the variable ?SOMEONE that are consistent with the information in the database.

```
[?SOMEONE MATTHEW]
[?SOMEONE JOSHUA]
NO (MORE) ANSWERS
```

We can have multiple variables. For example:

```
WHICH [KILLED ?MAN ?IMPLEMENT]
```

will give the answer:

```
[?MAN ZACHARIAH] [?IMPLEMENT AXE]
NO (MORE) ANSWERS
```

Let's consider the procedures that enable this analysis of the database, individually. WHICH passes the job over to FIND, indicating DATABASE as the source of facts.

```
TO WHICH :QUERY
   FIND :QUERY :DATABASE
   PRINT [NO (MORE) ANSWERS]
END
```

FIND sets up two global variables, VARS and ANS: VARS is used to hold each possible set of values of the variables in the question, and these are collected together in the list ANS.

```
TO FIND :QUERIES :DATA
   MAKE "VARS []
   MAKE "ANS []
   COMPARE :QUERY :DATA
   PRINTL :ANS
END
```

COMPARE looks at each fact in the database in turn. If there is a match then the new set of values in VARS are added to ANS before setting VARS back to the empty list. COMPARE then continues working through the DATABASE to see if there are any other possible matches.

```
TO COMPARE :QUERY :DATA
  IF EMPTY? :DATA THEN STOP
  IF MATCH? :QUERY FIRST :DATA THEN MAKE
    "ANS FPUT :VARS :ANS
  MAKE "VARS []
  COMPARE :QUERY BUTFIRST :DATA
END
```

To see what MATCH? does, consider the case where the inputs are [OWNS ?SOMEONE AXE] and [OWNS JOSHUA AXE] in response to which MATCH? outputs TRUE and sets VARS to [?SOMEONE JOSHUA]. If the inputs are [OWNS ?SOMEONE AXE] and [KILLED ZACHARIAH AXE], then MATCH? outputs FALSE.

The real difficulties arise, however, if there is more than one variable involved. VALUE? is used to check if the variable has already been assigned a value for that fact in the database.

We have used here an alternative notation for conditionals in LOGO. TEST evaluates a condition. If the result is true then the actions following IFTRUE will be performed, otherwise the actions following IFFALSE will be carried out.

```
TO MATCH? :QUERY :FACT
  IF ALLOF EMPTY? :QUERY EMPTY? :FACT THEN
    OUTPUT "TRUE
  TEST FIRST FIRST :QUERY = "?
  IFTRUE IF NOT VALUE? FIRST :QUERY FIRST
    :FACT :VARS THEN OUTPUT "FALSE
  IFFALSE IF NOT ( FIRST :QUERY = FIRST :FACT )
    THEN OUTPUT "FALSE
  OUTPUT MATCH? BUTFIRST :QUERY BUTFIRST
    :FACT
END
```

To see how VALUE? works, let's first consider the case where the inputs are ?IMPLEMENT, AXE, and [?MAN ZACHARIAH]. VALUE? tries to ascertain whether the variable ?IMPLEMENT could have the value AXE. There are three possibilities: ?IMPLEMENT already has a value, which is not AXE, and VALUE? outputs FALSE; ?IMPLEMENT already has the value AXE, and VALUE outputs TRUE; or ?IMPLEMENT does not have a value, so it is given the value AXE, and this information is added to VARS and TRUE is output.

```
TO VALUE? :NAME :VALUE :VLIST
  IF EMPTY? :VLIST THEN MAKE "VARS LPUT LIST
    :NAME :VALUE :VARS OUTPUT "TRUE
  TEST :NAME = FIRST FIRST :VLIST
  IFTRUE IF :VALUE = LAST FIRST :VLIST THEN
    OUTPUT "TRUE ELSE OUTPUT "FALSE
  OUTPUT VALUE? :NAME :VALUE BUTFIRST
    :VLIST
END
```

PRINTL simply arranges for the components of ANS to be printed out below each other.

```
TO PRINTL :LIST
  IF EMPTY? :LIST STOP
  PRINT FIRST :LIST
  PRINTL BUTFIRST :LIST
END
```

MORE COMPLEX ENQUIRIES

Our investigation will not go far, however, unless we can ask more complex questions, such as 'What implement killed Zachariah, and who owns such an implement?' In LOGO, this reads:

```
WHICH [[KILLED ZACHARIAH ?IMPLEMENT]
  [OWNS ?SUSPECT ?IMPLEMENT]]
```

WHICH now takes a list of queries as input and the values found will be those that make all of the queries true. If you then wish to ask a single query with this new form of WHICH the syntax we use is:

```
WHICH [[OWNS ?ANY KNIFE]]
```

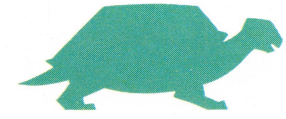
We need make only minor alterations to these procedures:

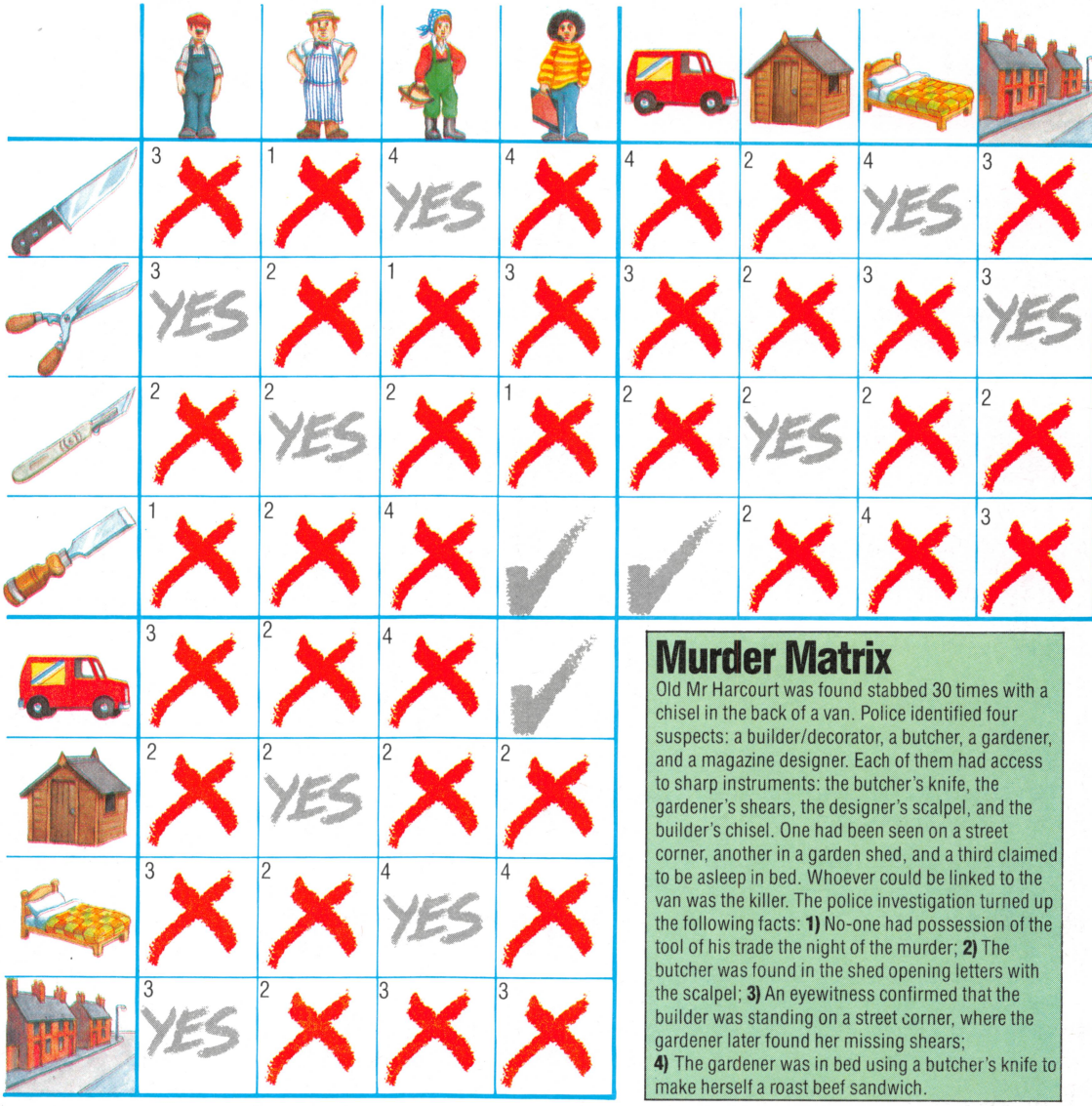
```
TO WHICH :QUERIES
  FIND :QUERIES :DATABASE
  PRINT [NO (MORE) ANSWERS]
END

TO FIND :QUERIES :DATA
  MAKE "VARS []
  MAKE "ANS []
  COMPARE :QUERIES :DATA
  PRINTL :ANS
END
```

COMPARE now has a rather difficult job to do. Let's take [[KILLED ZACHARIAH ?IMPLEMENT][OWNS ?SUSPECT ?IMPLEMENT]] as an example input. COMPARE goes through the database, one fact at a time, to find a match for the first query, and ends up matching ?IMPLEMENT with AXE. The routine then considers the second query ([OWNS ?SUSPECT ?IMPLEMENT]), starting again from the beginning of the database. A match is found for the second condition, with the value of ?IMPLEMENT as AXE and ?SUSPECT as MATTHEW. There are no more queries, so this is a possible solution.

But we have not finished yet; there may be other values that satisfy the second query, while keeping ?IMPLEMENT as AXE. So COMPARE now proceeds through the database from the point it left off, and indeed finds a second solution with ?SUSPECT as JOSHUA. Of course, the procedure does not stop there, but continues searching the DATABASE. This time it reaches the end without finding any new matching values.

It is possible, however, that there is another solution to the first query — other than ?IMPLEMENT as AXE — so we must go back to the point where we found that match in the database and carry on from there. This process is called *backtracking*. In this case, there are in fact no other solutions.





DAVID HIGHAM

Murder Matrix

Old Mr Harcourt was found stabbed 30 times with a chisel in the back of a van. Police identified four suspects: a builder/decorator, a butcher, a gardener, and a magazine designer. Each of them had access to sharp instruments: the butcher's knife, the gardener's shears, the designer's scalpel, and the builder's chisel. One had been seen on a street corner, another in a garden shed, and a third claimed to be asleep in bed. Whoever could be linked to the van was the killer. The police investigation turned up the following facts: **1)** No-one had possession of the tool of his trade the night of the murder; **2)** The butcher was found in the shed opening letters with the scalpel; **3)** An eyewitness confirmed that the builder was standing on a street corner, where the gardener later found her missing shears; **4)** The gardener was in bed using a butcher's knife to make herself a roast beef sandwich.

Logo Flavours

Some versions of MIT LOGO do not have EMPTY? or MEMBER?. Definitions for these have been given previously (see pages 754 and 776).

In all LCSI versions use EMPTYP for EMPTY? and MEMBERP for MEMBER?. There is a primitive, EQUALP, which tests whether its two inputs are the same. Use it for comparing lists and words instead of an equals sign (which works for lists on some LCSI versions, but not on others).

The IF syntax in LCSI LOGO is demonstrated by:

```
IF EMPTYP :CONTENTS
[PRINT [NOTHING
 SPECIAL]] [PRINT
 :CONTENTS]
```

The first list after the condition is performed if the condition is true, and the second if it is false.

LCSI LOGO also supports the TEST, IFTRUE, IFFALSE syntax for conditionals

In order not to lose track of where it is up to in its assignment of variables, COMPARE puts the present values on a *stack* before MATCH? is used (since MATCH? may alter these assignments), and then recovers these values afterwards. Here is the full procedure:

```
TO COMPARE :QUERIES :DATA
  IF EMPTY? :QUERIES THEN MAKE "ANS FPUT
    :VARS :ANS STOP
  IF EMPTY? :DATA THEN STOP
  PUSH :VARS
  TEST MATCH? FIRST :QUERIES FIRST :DATA
  IFTRUE COMPARE BUTFIRST :QUERIES
    :DATABASE
  PULL "VARS
  COMPARE :QUERIES BUTFIRST :DATA
END
```

In COMPARE we use a stack to keep track of the value of VARS, instead of using a temporary variable, because COMPARE could call itself between the time we want to save the values and the time we want to restore them. Therefore, any such temporary variable could be overwritten by the next call and the original values lost. The stack prevents this from happening.

PUSH puts a value on 'top' of the stack, first creating the variable STACK if it does not already exist.

```
TO PUSH :DATA
  IF NOT THING? :STACK THEN MAKE "STACK []
  MAKE "STACK FPUT :DATA :STACK
END
```

PULL takes an item from the stack, and assigns it as the value of a variable.

```
TO PULL :NAME
  MAKE :NAME FIRST :STACK
  MAKE "STACK BUTFIRST :STACK
END
```

What we have then are the rudiments of a 'logic programming' language. That is a language in which we simply add facts and rules to a database and then query that database by means of logical descriptions of the data we require. The best example to date of a logic programming language is PROLOG — but that's another story!

OUTBOARD MOTOR

A special type of electric motor called a 'stepper' motor is used to power the Workshop robot. Stepper motors are favoured for computer control because they use logic signals to control their speed and rotation through discrete steps and are thus ideally suited to digital control.

The construction of a stepper motor is very different from that of a normal motor. To understand the principles of operation we shall consider how a simplified stepper motor works. In our example (see the diagram headed 'One Step At A Time') there are two set of windings ('a' and 'b') on the stator and two pairs of electromagnetic poles on the rotor. In the motors that are used in the Workshop robot there are more stator windings and more rotor poles than our example shows.

The only problem with this convenient form of motor control is that the motor consumes as much current when stationary as it does when in motion. In addition, it cannot be rotated at high speed — the different coils cannot energise and de-energise quickly enough. However neither of these problems is significant in our robot application.

Our simplified motor is capable of turning in steps of 45° only. Additionally, the direction of rotation cannot be controlled. The motors used in the robot, however, have four sets of coils that are energised in pairs, and the rotor also has many more coils than our example shows. This means that the direction of rotation may be controlled and that the step angle is reduced to 7.5°. To achieve this accurate stepwise rotation, the four coils must be energised in a particular and complicated sequence as follows:

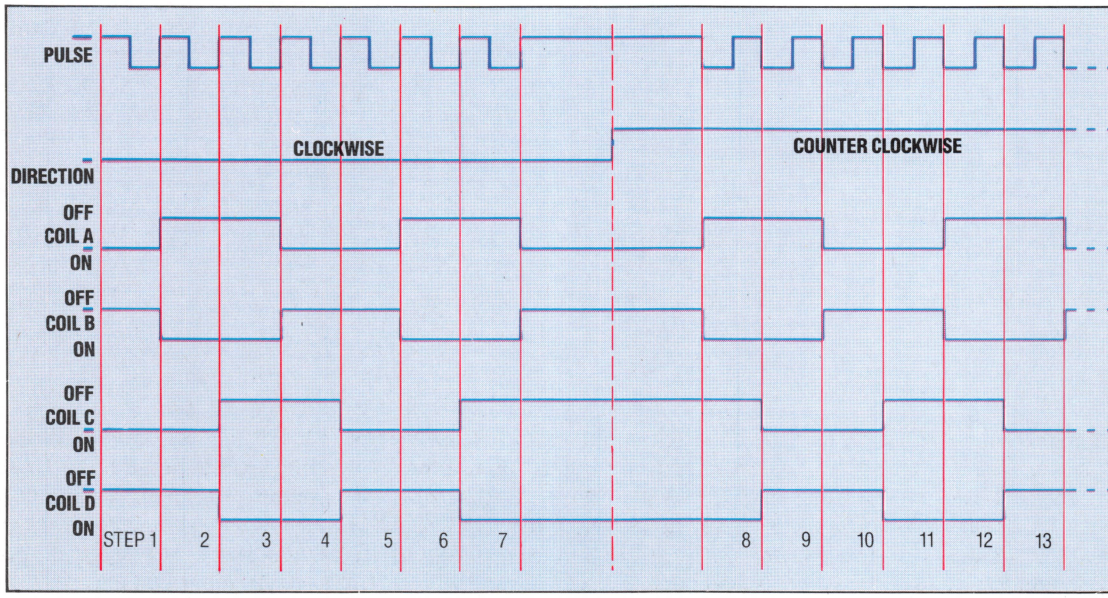
Stator Coils Energising Sequence Table

Step	Coil A	Coil B	Coil C	Coil D	
1	on	off	on	off	
2	off	on	on	off	
3	off	on	off	on	
4	on	off	off	on	
5	on	off	on	off	etc

This sequence of energisation could be provided by software, using four bits of the user port to control the four coils. However, this requires some complicated programming and BASIC could certainly not produce these control sequences quickly enough. A simpler method is to use a chip that has been specially designed for the control of stepper motors — the SAA 1027. This contains the output drivers and all the logic circuits to energise the coils in the correct order to drive a stepper motor.

To rotate the motor through one step, a single pulse from the user port is required, with a further signal line being needed to determine the direction of rotation. The chip contains input stages to detect the changes in the three inputs: a pulse to rotate the motor one more step, a reset input, and a direction input that reverses the stator coils energising sequence. The inputs are fed into a bi-directional counter circuit to produce the correct output sequence to the stator coils.

Finally, the chip also contains a power output driver stage that can handle up to 500mW. The inclusion of this stage means that the motor can be



Energising The Coils

The diagram shows how the four coils are energised in sequence producing a clockwise movement in the rotor. After steps 1, 2, 3 and 4 have been completed the sequence restarts; step 5 energising the same combination of coils as step 1. When the direction signal changes the sequence is reversed. Step 9 energises the coils in the same combination as step 7, step 10 energises the coils in the same way as step 6, and so on, causing the rotor to turn anti-clockwise, the rotor position after step 8 being the most clockwise position reached

connected directly to the chip without the need for external power transistors.

The complexity of the stepper motor driver chip means that the rest of the circuit needed for the robot control is very simple indeed. Each motor requires one of these chips, to which the motor is connected. Unfortunately, the driver chips operate at a voltage of about 12 volts, while your computer user port operates at five volts. That is, a logic zero is zero volts (or thereabouts) and a one is five volts. The driver chip inputs require zero volts for a zero input and between 7.5 and 12 volts for a one. To interface the user port to the driver chips we therefore also need a special two-voltage buffer chip with the inputs operating on one voltage and the outputs on another. This is the 40109 chip that is also needed in the circuit.

Parts List

MAPLIN

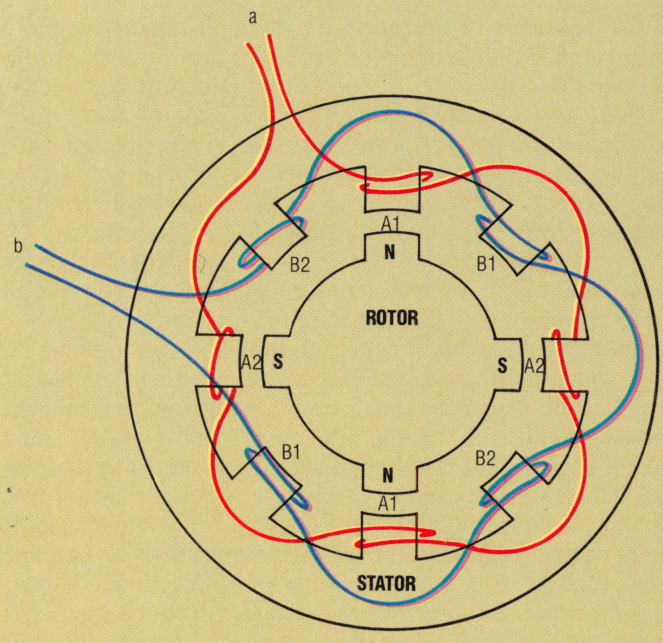
No	Item	Source
1	40109 buffer chip	QW67X
3	16-pin DIL sockets	BL19V
2	100 ohm resistors	M100R
2	270 ohm 0.5 watt resistors	S270R
2	0.1 µ F capacitors	YR75S
1	1000 µ F 25v capacitor	FB83E
1	24 strip x 50 hole veroboard	FL07H
1	reel tinned 20 swg wire	BL13P

RADIO SPARES

2	SAA 1027 stepper motor drivers	300-237

One Step At A Time

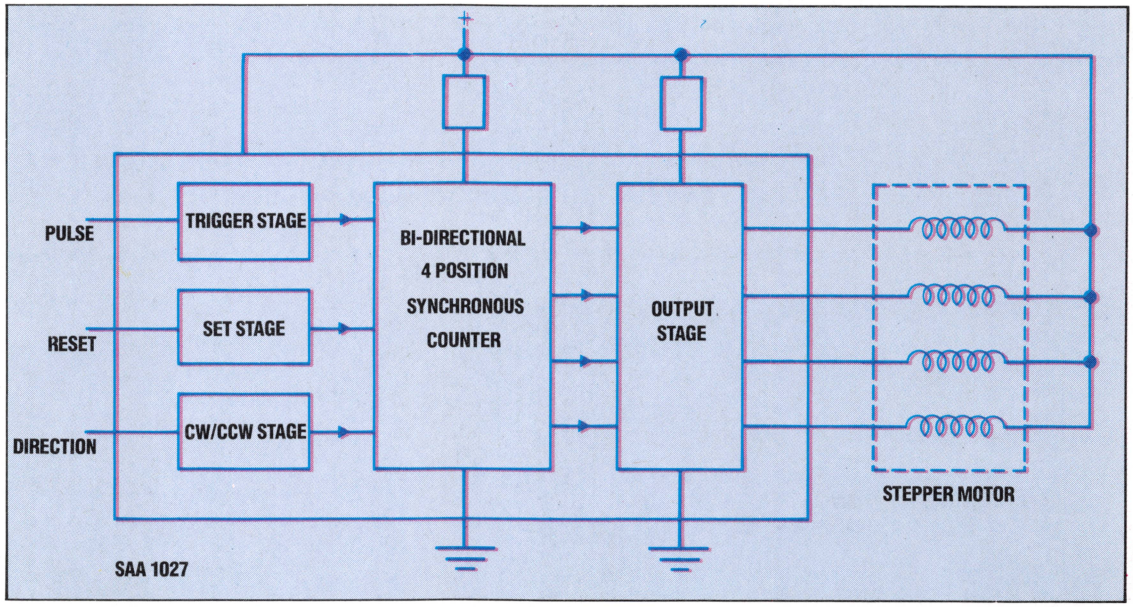
This simplified diagram of a stepper motor shows two stator winding circuits 'a' and 'b' and a rotor. The rotor is forced to rotate in a clockwise direction by alternate energising of the two winding circuits. Notice that the coil pairs A1 and A2 are wound in opposite directions. When winding 'a' is energised inducing south poles in pair A1, north poles are therefore induced in pair A2. Coil pairs B1 and B2 are similarly wound in opposing directions. The minimum step angle that this simple motor can turn through is 45°. The motors used in the Workshop robot have more windings on the stator and an increased number of poles on the rotor, allowing these motors to turn through 7.5° steps



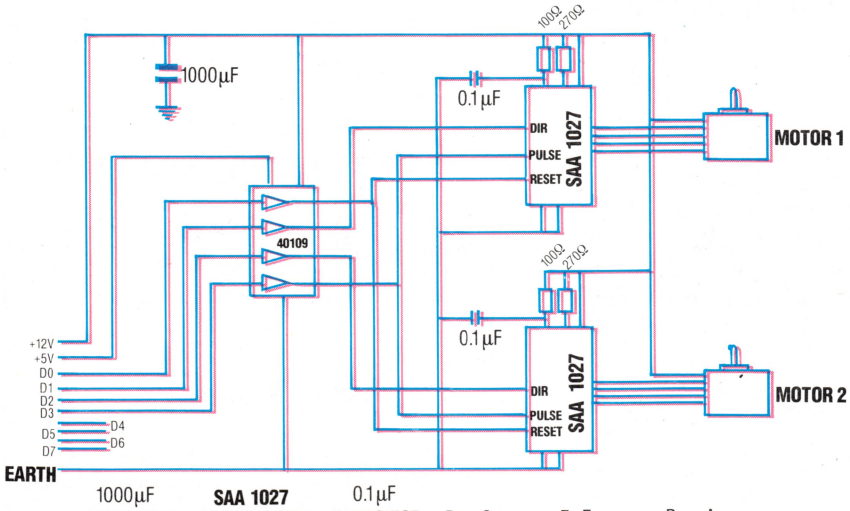
KEVIN JONES

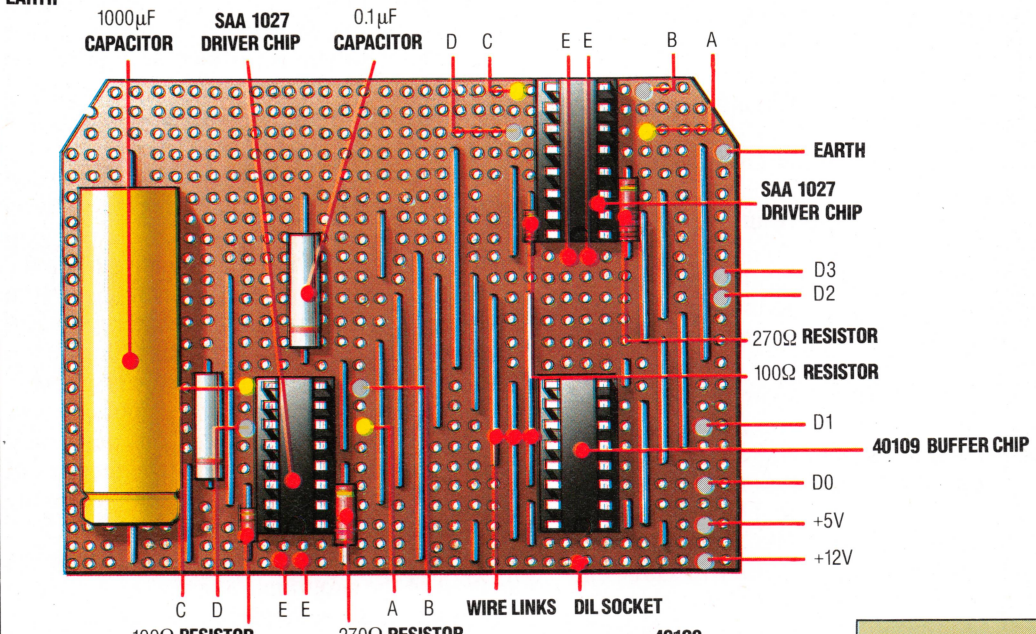
The Driving Force

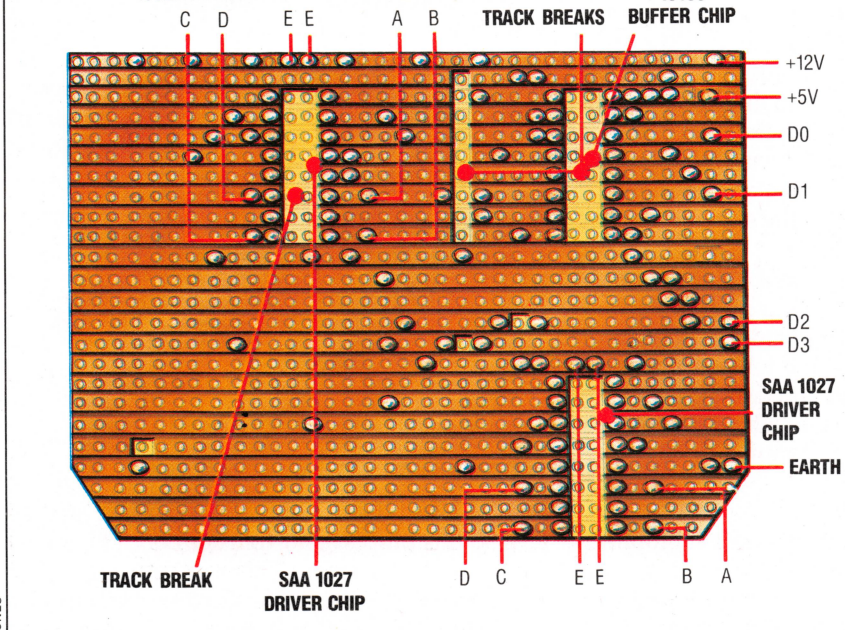
Although the logic of the stepper motor driving chips is complex, the principles of operation are easily understood. In order to turn the rotor the stator coils must be energised in a certain sequence. A bi-directional counter moves through this sequence a stage at a time in response to a pulse signal. The sequence can also be stepped through in the opposite direction if the direction line input is changed, causing the rotor to turn in the opposite direction. A third input allows the rotor to be reset to its position at the beginning of the sequence, if required



LIZ DIXON







KEVIN JONES

The Circuit Diagram
The circuitry required to drive the two stepper motors is straightforward, two SAA 1027 driver chips being used to provide the correct coil energising sequence to each motor. As the driver chips operate at 12v and user port signals from your micro are only 5v, an additional buffer chip is used to isolate the 12v circuitry from the computer and translate the low voltage user port signals into the higher voltage signals needed by the driver chips. In the next instalment we shall show how the circuit board connects to the motors and D plug and how to make the correct connections with the computer's user port

The Project: Building The Board

- Cut the board to size (24 strips x 35 holes) and make the track cuts as shown in the diagram.
- Solder in the three chip sockets first and then all the wire links.
- Solder in the four resistors. It does not matter which way around these are fitted onto the board.
- The two 0.1µF capacitors can also be fitted in either orientation, but the large 1000µF smoothing capacitor must be fitted with the positive terminal at the side of the board with the two chips.
- Now fit the chips in position, making sure that they are the right way around. The end of the chip with a notch in it should be at the side of the board with the two chip sockets.
- Check the whole board very carefully. Check not only that all the components are correctly positioned, but also that there are no blobs of solder bridging across adjacent tracks. It is a good idea to run a sharp knife along the gap between each track to clear away any solder debris.

BYTE THE DUST

Our 6809 Assembly language course comes to its conclusion with this instalment. We tie up all the loose ends of our debugging program, provide an overall view of the flow of command within it, and finally code the main module itself.

The first task of the main module is to set up the interrupt mechanism, which allows us to set breakpoints in the program being debugged. These transfer control to the debugger and allow us to inspect the contents of the registers and memory locations. We must then obtain the starting address of the program being debugged so that control can be passed to it using the S command. The rest of the main routine involves getting commands from the keyboard and executing them; control is transferred to the program being debugged by the S and G commands and returned to the debugger by the SWI instructions inserted at the breakpoints.

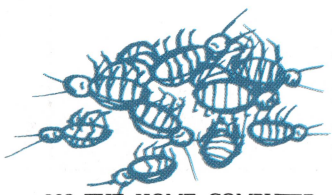
Two stages of initialisation for this module were coded in the last instalment (see page 817). The entry point for interrupts comes immediately after the call to this subroutine. The first instruction here is to save the stack pointer, S so that it can be used to reference the values from the registers saved on the stack by the SWI. The next stage is command interpretation. We have already developed subroutines to perform all the commands, so the problem here is to select the subroutine appropriate to the command entered.

It is possible to code this as a set of nested IF statements, but we will use the fact that the Get-Command routine returns an offset into a table of command characters to perform these calls using a jump table. This is not perhaps the most efficient method in this instance, but it is a useful technique that is worth looking at. It involves setting up a table of addresses for each of the subroutines that actually carry out a command.

The JMP instruction, unlike the branch instructions, can use any of the normal addressing modes, including indexed and indirect. If we load X with the base address of the table and use the offset in B (doubled because this will be a table of 16-bit addresses, unlike the table of eight-bit command letters), then the command:

```
JMP [B,X]
```

will transfer control to the appropriate subroutine. The BSR call is made to the address of this jump instruction. As we need to set up this table in advance, it is necessary to have another stage of initialisation to carry out this operation.



PROCESS SET-UP-JUMP-TABLE

Data:

Jump-Table is a table of eight 16-bit addresses
CMDB, CMDU, etc. are the start addresses for the subroutines

Process:

```
For each subroutine
    Get start address
    Save start address in Jump-Table
Endfor
```

We must now consider what is to happen at the end of the run, when the quit command (Q) is issued, although there is, in fact, very little that needs doing. It makes sense to leave both the debugger and the program intact so that they can be re-entered if necessary.

The stack should be in the same situation when we exit as it was when we started. One solution would be to use a separate stack for our program by setting S to a new value and then restoring the old value. This is often a useful technique, but in our situation it may be difficult finding unused space in memory, with the debugger sitting on top of another program. Another solution is simply to increment S by the appropriate amount to lose anything that we have left there, but this is also difficult because we do not know whether or not an interrupt has occurred and the amounts on the stack will be different. The simplest solution is to save the initial value of S and restore it as the last operation of the program.

The interrupt mechanism, as set up in the initialisation procedure, stores three bytes at the address given in the SWI vector at $FFFA; we must restore this or strange results may occur if the operating system uses SWI for its own purposes. What we clearly need is a further stage of initialisation where we save these values to be restored in our quit routine.

PROCESS SAVE-VALUES

Data:

Saved is the five bytes to store the saved values
Stack-Pointer is the current value of S, plus two
SWI-Vector is found at $FFFA

Process:

```
Save Stack-Pointer in Saved
Get SWI-Vector
Save three bytes at SWI-Vector in Saved
```

The quit routine (command Q) must simply reverse this process and transfer control back to the operating system. This can be done in a number of ways: the SWI instruction itself can be used, after its vector has been reset, or a jump can be made to a known entry point in the operating

system. A jump via the reset vector that resides at $FFFE is guaranteed to return control to the operating system, though it may cause a cold start.

PROCESS QUIT

Data:

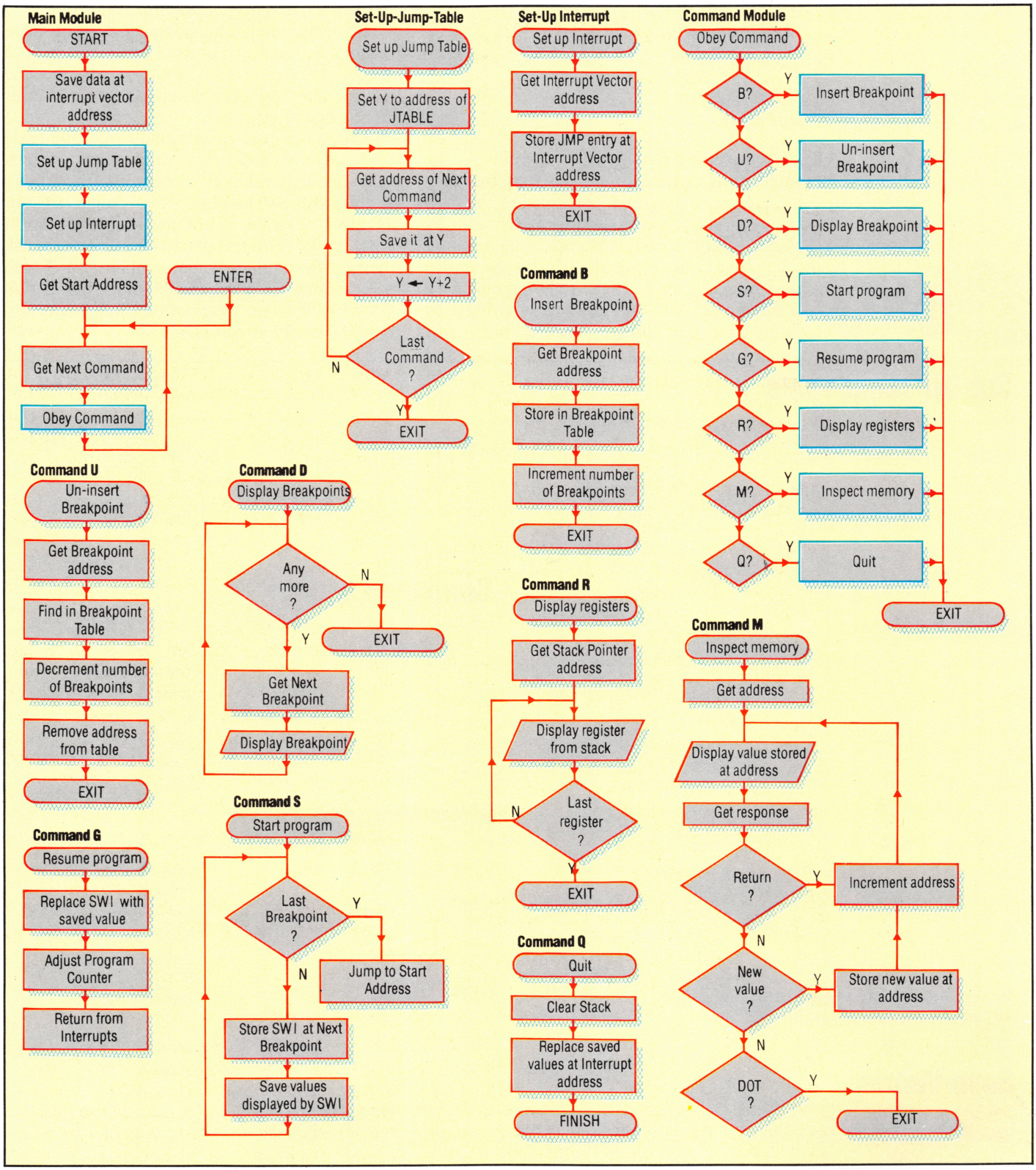
Saved is the five bytes to store the saved values
Stack-Pointer is the current value of S, plus two
SWI-Vector is at $FFFA
Reset-Vector is at $FFFE

Process:

Restore three bytes from Saved at SWI-Vector
Restore Stack-Pointer
Jump to operating system

We are now ready to code the main module. The design has altered slightly from when we first sketched it out, but it remains essentially the same.

Program Flow
These flow diagrams correspond to the debugger program modules. They are placed in the order in which they are called by other routines. Within the diagrams, boxes coloured blue indicate separate routines being called



LIZ DIXON

THE MAIN MODULE

Data:

Prompt for command entry is ASCII character '>'
Command-Offset into table of command characters and Jump-Table

Process:

Save-Values
Set-Up-Jump-Table
Set-Up-Interrupt
Get Start-Address
Repeat
  Display Prompt
  Get-Command
  Do-Command
Indefinitely

That completes our debugger program. At the moment it is rather fragmented, but that is a consequence of modular construction. At this point we can optimise the code if we wish by looking for short cuts. For example, you may find that you have had to move a lot of values around to make sure that they are in the right registers for a subroutine, so you might make savings by redefining register usage. This is not really advisable unless memory space is very restricted. We have defined the same data areas in a number of different places, as they are required. There are two ways in which you might handle data areas in the complete program: you can retain the data with the module that uses it, which is theoretically the best option; or you can define all the data together at the start of the program, which has real advantages if you ever want to use a disassembler (or even a debugger) on the program.

The debugger should be loaded into any spare memory not occupied or used by the program to be debugged. It is entered by making a jump to the DEBUG entry point, so it is necessary to know this address before you start.

In the later part of this 6809 machine code series, we have tried to show the best way in which programs are developed, illustrated with a variety of techniques. Therefore, the design of our debugger program is not necessarily the most efficient way to do this particular job. If you have followed everything, however, then you should have a fairly comprehensive understanding of Assembly language programming in general, and 6809 Assembly in particular.

Set-Up-Jump-Table

```
JTABLE  RMB   16          Space for 8 two-byte addresses
SETUPJ  LEAY  JTABLE,PCR  Base address of table in Y
        LEAX  CMDB,PCR    Start address of CMDB subroutine
        STX   ,Y++        Store it in table
        LEAX  CMDU,PCR    Start address of CMDU subroutine
        STX   ,Y++        Store it in table
        LEAX  CMDD,PCR    Start address of CMDD subroutine
        STX   ,Y++        Store it in table
        LEAX  CMDS,PCR    Start address of CMDS subroutine
        STX   ,Y++        Store it in table
        LEAX  CMDG,PCR    Start address of CMDG subroutine
        STX   ,Y++        Store it in table
        LEAX  CMDR,PCR    Start address of CMDR subroutine
        STX   ,Y++        Store it in table
        LEAX  CMDM,PCR    Start address of CMDM subroutine
        STX   ,Y++        Store it in table
        LEAX  CMDQ,PCR    Start address of CMDQ subroutine
        STX   ,Y++        Store it in table
```

This is the actual jump to the subroutine. We assume that X contains the address of JTABLE and B the offset

```
DOCMD   JMP   [B,X]
```

Save-Values

```
SAVED   RMB   5           Five bytes to be saved
SAVEIT  LEAX  SAVED,PCR   Get address to save in
        TFR   S,D         Move S to D
        ADDD  #2          Add two to take care of the return address
        STD   ,X++        Save it
        LDY   $FFFA       Get Interrupt vector address
        LDA   ,Y+         Get first byte to be saved
        STA   ,X+         Save it
        LDD   ,Y          Get other two bytes
        STD   ,X          Save them
        RTS
```

Command Q

```
CMDQ    LEAX  SAVED,PCR   Address of Saved
        LDY   $FFFA       SWI-Vector
        LDA   2,X         First of three bytes
        STA   ,Y+         Restored
        LDD   3,X         Other two bytes
        STD   ,Y          Restored
        LDS   ,X          Saved Stack-Pointer
        JMP   [$FFFE]     Indirect jump via reset vector
```

Main Module

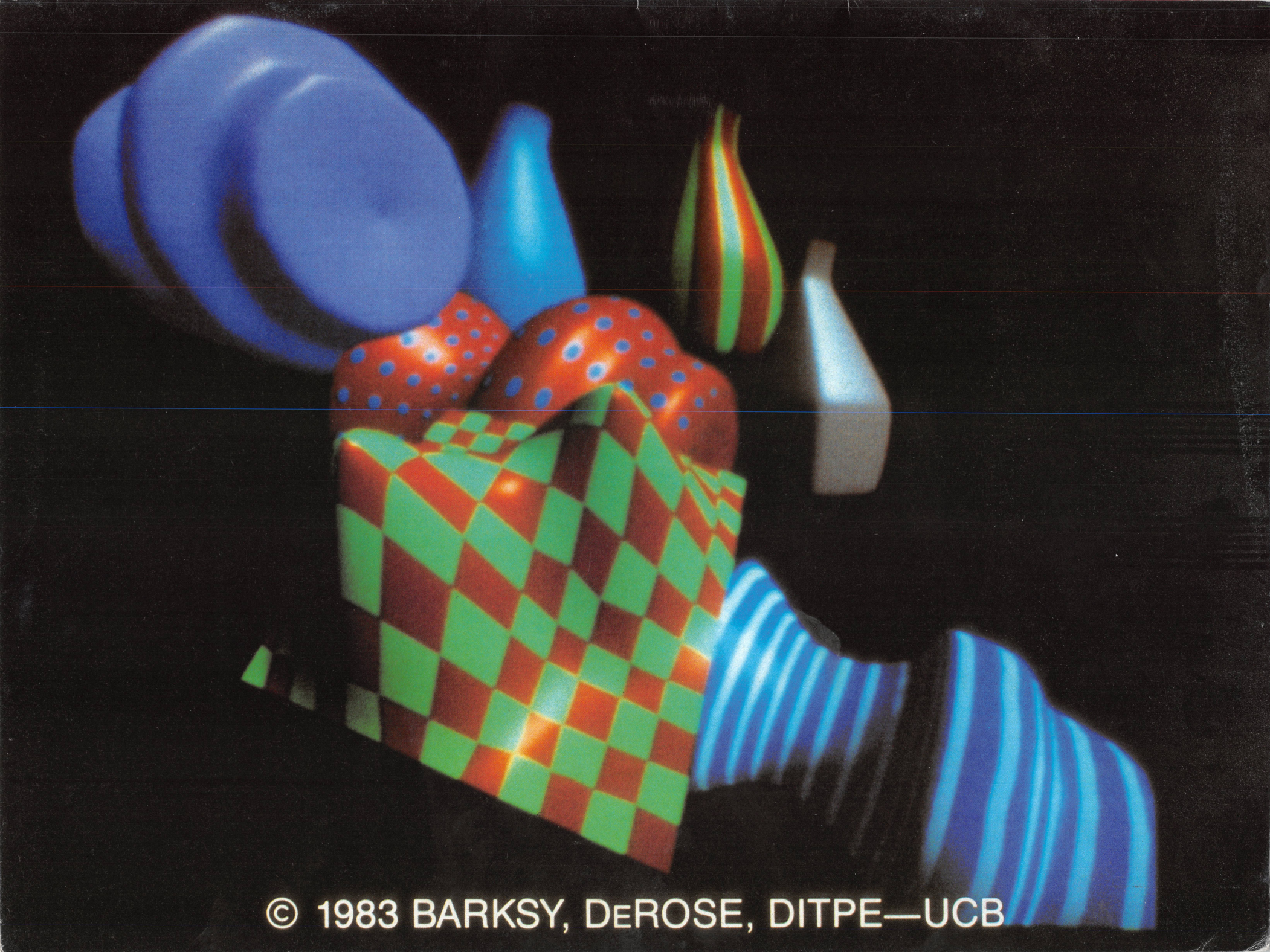
```
PROMPT  FCB   '>
STACKP  RMB   2           Stack-Pointer for Display-Registers
DEBUG   BSR   SAVEIT      Save-Values
        BSR   SETUPJ      Set-Up-Jump-Table
        BSR   INIT        Set-Up-Interrupt and Get Start-Address
ENTRY   STS   STACKP,PCR  Save Stack-Pointer
        LEAX  JTABLE,PCR
REPT02  LDA   PROMPT,PCR  Get prompt and
        BSR   OUTCH       display it
        BSR   GETCOM      Get Command
        LSLB              Double offset for 16-bit table
        BSR   DOCMD       Obey Command
        BRA   REPT02      Next Command
```

DATABASE

Here, courtesy of Oric Products International, we publish the first instalment of the 6502 programmers' reference card

MNEMONIC	DESCRIPTION	R1	R2	IMPLIED	IMMED	ZERO PAGE	Z-PAGE X
ADC	ADD WITH CARRY	A			69	65	75
AND	LOGICAL AND	A			29	25	35
ASL	ARITHMETIC SHIFT LEFT	A/M		0A(A)		06	16
BCC	BRANCH ON CARRY CLEAR						
BCS	BRANCH ON CARRY SET						
BEQ	BRANCH IF EQUAL TO ZERO						
BIT	COMPARE BITS WITH ACCUMULATOR					24	
BMI	BRANCH ON MINUS						
BNE	BRANCH ON NOT EQUAL TO ZERO						
BPL	BRANCH ON PLUS						
BRK	BREAK	S		00			
BVC	BRANCH ON OVERFLOW CLEAR						
BVS	BRANCH ON OVERFLOW SET						
CLC	CLEAR CARRY			18			
CLD	CLEAR DECIMAL			D8			
CLI	CLEAR INTERRUPT MASK			58			
CLV	CLEAR OVERFLOW FLAG			B8			
CMP	COMPARE TO ACCUMULATOR				C9	C5	D5
CPX	COMPARE TO REG-X				E0	E4	
CPY	COMPARE TO REG-Y				C0	C4	
DEC	DECREMENT MEMORY					C6	D6
DEX	DECREMENT REG-X	X		CA			
DEY	DECREMENT REG-Y	Y		88			
EOR	EXCLUSIVE OR ACCUMULATOR	A			49	45	55
INC	INCREMENT MEMORY					E6	F6
INX	INCREMENT REG-X	X		E8			
INY	INCREMENT REG-Y	Y		C8			
JMP	JUMP TO ADDRESS	PC					

Z-PAGE Y	INDIRECT X	INDIRECT Y	RELATIVE ADDRESS	ABSOLUTE ADDRESS	ABSOLUTE X	ABSOLUTE Y	INDIRECT	P-REGISTER N	V	*	B	D	I	Z	C
	61	71		6D	7D	79		×	×					×	×
	21	31		2D	3D	39		×						×	
				0E	1E			×						×	×
			90												
			B0												
			F0												
				2C				M7	M6					×	
			30												
			D0												
			10												
											×				
			50												
			70												
															0
												0			
													0		
									0						
	C1	D1		CD	DD	D9		×						×	×
				EC				×						×	×
				CC				×						×	×
				CE	DE			×						×	
								×						×	
								×						×	
	41	51		4D	5D	59		×						×	
				EE	FE			×						×	
								×						×	
								×						×	
				4C			6C								


© 1983 BARKSY, DEROSE, DITPE—UCB